

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

C# i .NET

Autor: Stephen C. Perry

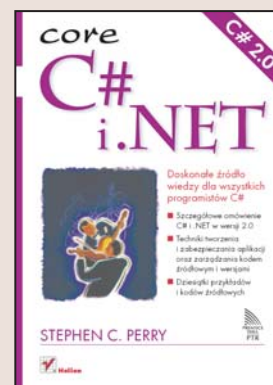
Tłumaczenie: Mikołaj Szczepaniak (przedmowa, rozdz.

1 – 10), Tomasz Walczak (rozdz. 11 – 18, dod. A, B)

ISBN: 83-246-0320-4

Tytuł oryginału: [Core C# and .NET](#)

Format: B5, stron: 912



Doskonałe źródło wiedzy dla wszystkich programistów C#

- Szczegółowe omówienie C# i .NET w wersji 2.0
- Techniki tworzenia i zabezpieczania aplikacji oraz zarządzania kodem źródłowym i wersjami
- Dziesiątki przykładów i kodów źródłowych

Platforma .NET i język C# spowodowały prawdziwą rewolucję na rynku narzędzi programistycznych. Wersja 2.0 przyniosła kilka interesujących rozwiązań, w tym nowe typy danych, komponenty i klasy. Dodatkowe funkcje języka C# pozwalają na przyspieszenie i usprawnienie procesu tworzenia aplikacji oraz jej wdrażania i rozwijania. Dzięki platformie .NET przygotowanie wydajnej, bezpiecznej i łatwej do rozbudowy aplikacji stało się znacznie prostsze i szybsze.

Książka „C# i .NET” przedstawia możliwości platformy .NET i napisanego specjalnie dla niej języka C#. Opisuje elementy platformy, składnię i możliwości języka C# oraz zagadnienia związane z tworzeniem programów za pomocą tych narzędzi. Omawia zasady pisania aplikacji Window Forms i efektywne techniki stosowania klas graficznych z biblioteki GDI+. Charakteryzuje także technologię ADO.NET, metody korzystania z plików XML, przetwarzanie wielowątkowe oraz techniki tworzenia wydajnych aplikacji internetowych w technologii ASP.NET. Szczególną uwagę poświęcono bezpieczeństwu programów i danych oraz skalowaniu i lokalizacji oprogramowania.

- Elementy platformy .NET
- Podstawy języka C#
- Praca z klasami i obiektami w C#
- Przetwarzanie tekstu oraz operacje wejścia i wyjścia
- Tworzenie aplikacji Window Forms
- Korzystanie z biblioteki GDI+
- Połączenia z bazami danych za pomocą ADO.NET
- Aplikacje wielowątkowe oraz rozproszone
- Tworzenie aplikacji internetowych

Jeśli chcesz napisać w C# aplikację dobrej jakości, sięgnij po tę książkę



Spis treści

O autorze	17
Podziękowania	18
Słowo wstępne	19
Przedmowa	21
Część I Podstawy programowania w języku C# i wprowadzenie do technologii .NET	25
Rozdział 1. Wprowadzenie do technologii .NET i języka C#	27
1.1. Przegląd składników platformy .NET	28
Microsoft .NET i standardy CLI	30
1.2. Wspólne środowisko uruchomieniowe (CLR)	32
Kompilacja kodu platformy .NET	32
Wspólny system typów (CTS)	34
Zestawy .NET	36
1.3. Biblioteka klas platformy (FCL)	41
1.4. Praca z platformą .NET i zestawem narzędzi .NET Framework SDK	44
Aktualizowanie platformy .NET Framework	45
Narzędzia platformy .NET Framework	46
Ildasm.exe	47
wincv.exe	50
Narzędzie Framework Configuration	50
1.5. Wyjaśnienie działania kompilatora C#	53
Lokalizowanie kompilatora	53
Kompilowanie oprogramowania z poziomu wiersza poleceń	54
1.6. Podsumowanie	57
1.7. Sprawdź, czego się nauczyłeś	57
Rozdział 2. Podstawy języka C#	59
2.1. Podstawowa struktura programu C#	60
Ogólne uwagi odnośnie programowania w języku C#	62
2.2. Typy proste	65
decimal	67
bool	67

char	67
byte, sbyte	68
short, int, long	68
single i double	68
Konwertowanie łańcuchów numerycznych za pomocą metod Parse() i TryParse()	69
2.3. Operatory arytmetyczne, logiczne i warunkowe	69
Operatory arytmetyczne	69
Operatory warunkowe i relacyjne	70
Instrukcje kontrolujące przepływ sterowania	71
if-else	71
switch	73
2.4. Pętle	74
Pętla while	74
Pętla do	74
Pętla for	75
Pętla foreach	75
Kontrola przepływu sterowania w pętlach	76
2.5. Dyrektywy preprocesora C#	76
Kompilacja warunkowa	77
Dyrektywy diagnostyczne	79
Regiony (obszary) kodu	79
2.6. Łańcuchy znaków	79
Stałe łańcuchowe	80
Operacje na łańcuchach	81
2.7. Typy wyliczeniowe	84
Praca z typami wyliczeniowymi	84
Metody klasy System.Enum	85
Typy wyliczeniowe i flagi bitowe	86
2.8. Tablice	87
Deklarowanie i tworzenie tablic	88
Stosowanie metod i właściwości tablicy System.Array	88
2.9. Typy referencyjne i wartościowe	90
Klasy System.Object i System.ValueType	90
Techniki przydzielania pamięci dla typów referencyjnych i typów wartościowych	91
Technika pakowania	92
Podsumowanie różnic pomiędzy typami wartościowymi a typami referencyjnymi	94
2.10. Podsumowanie	95
2.11. Sprawdź, czego się nauczyłeś	95

Rozdział 3. Projektowanie klas w języku C# 97

3.1. Wprowadzenie do klas języka C#	98
3.2. Definiowanie klas	98
Atrybuty	99
Modyfikatory dostępu	102
Modyfikatory abstract, sealed i static	102
Identyfikator klasy	103
Klasy bazowe, interfejsy i dziedziczenie	103
3.3. Przegląd składowych klasy	104
Modyfikatory dostępu do składowych	104
3.4. Stałe, pola i właściwości	104
Stałe	106
Pola	107

Właściwości	109
Indeksery	111
3.5. Metody	113
Modyfikatory metod	113
Przekazywanie parametrów	118
3.6. Konstruktory	120
Konstruktor instancji	120
Konstruktor prywatny	124
Konstruktor statyczny	125
3.7. Delegacje i zdarzenia	126
Delegacje	127
Obsługa zdarzeń z wykorzystaniem delegacji	129
3.8. Przeciążanie operatorów	136
3.9. Interfejsy	139
Tworzenie i stosowanie interfejsów niestandardowych	139
Praca z interfejsami	142
3.10. Klasy uniwersalne	143
3.11. Struktury	146
Definiowanie struktur	147
Stosowanie struktur z metodami i właściwościami	148
3.12. Struktury kontra klasy	149
Struktury są typami wartościowymi, klasy są typami referencyjnymi	149
W przeciwieństwie do klas struktury nie mogą być dziedziczone	150
Ogólne reguły, które należy uwzględnić, wybierając pomiędzy strukturami a klasami	150
3.13. Podsumowanie	151
3.14. Sprawdź, czego się nauczyłeś	151
Rozdział 4. Praca z obiektami w języku C#	155
4.1. Tworzenie obiektów	156
Przykład: Tworzenie obiektów za pomocą wielu klas fabrykujących	158
4.2. Obsługa wyjątków	159
Klasa System.Exception	160
Pisanie kodu obsługującego wyjątki	160
Przykład: Obsługa wspólnych wyjątków SystemException	163
Tworzenie niestandardowych klas wyjątków	164
Wyjątki nieobsługiwane	167
Wskazówki dotyczące obsługi wyjątków	168
4.3. Implementowanie metod klasy System.Object w klasach niestandardowych	170
Metoda ToString() jako narzędzie opisywania obiektów	170
Metoda Equals() jako narzędzie porównywania obiektów	172
Klonowanie jako sposób tworzenia kopii obiektów	174
4.4. Praca z klasami i interfejsami kolekcji technologii .NET	176
Interfejsy kolekcji	176
Przestrzeń nazw System.Collections	185
Klasy Stack i Queue	185
Klasa ArrayList	187
Klasa Hashtable	188
Przestrzeń nazw System.Collections.Generic	191
4.5. Serializacja obiektów	195
Serializacja binarna	195

4.6. Zarządzanie cyklem życia obiektów	199
Odzyskiwanie pamięci w technologii .NET	199
4.7. Podsumowanie	205
4.8. Sprawdź, czego się nauczyłeś	205

Część II Tworzenie aplikacji z wykorzystaniem biblioteki klas platformy .NET Framework

207

Rozdział 5. Przetwarzanie tekstu i plikowe operacje wejścia-wyjścia w języku C# 209

5.1. Znaki i format Unicode	210
Format Unicode	211
Praca ze znakami	212
5.2. Klasa String	215
Tworzenie łańcuchów	216
Przegląd operacji na łańcuchach znakowych	218
5.3. Porównywanie łańcuchów znakowych	219
Stosowanie metody String.Compare()	219
Stosowanie metody String.CompareOrdinal()	221
5.4. Przeszukiwanie, modyfikowanie i kodowanie zawartości łańcuchów	222
Przeszukiwanie zawartości łańcuchów	222
Przeszukiwanie łańcuchów zawierających pary zastępcze	222
Przekształcanie łańcuchów	223
Kodowanie łańcuchów	225
5.5. Klasa StringBuilder	226
Przegląd składowych klasy StringBuilder	227
Konkatenacja w klasie StringBuilder kontra tradycyjna konkatenacja łańcuchów	228
5.6. Formatowanie wartości liczbowych oraz daty i godziny	229
Konstruowanie elementów formatowania	230
Formatowanie wartości liczbowych	231
Formatowanie daty i godziny	231
5.7. Wyrażenia regularne	236
Klasa Regex	237
Tworzenie wyrażeń regularnych	242
Przykład dopasowywania wzorca do łańcucha	242
Praca z grupami	245
Przykłady stosowania wyrażeń regularnych	247
5.8. Przestrzeń nazw System.IO — klasy obsługujące odczytywanie i zapisywanie strumieni danych	248
Klasa Stream	249
Klasa FileStream	249
Klasa MemoryStream	251
Klasa BufferedStream	252
Stosowanie klas StreamReader i StreamWriter odpowiednio do odczytywania i zapisywania wierszy tekstu	253
Klasy StringWriter i StringReader	255
Szyfrowanie danych za pomocą klasy CryptoStream	256
5.9. Przestrzeń nazw System.IO — katalogi i pliki	259
Klasa FileSystemInfo	259
Praca z katalogami za pomocą klas DirectoryInfo, Directory oraz Path	260
Praca z plikami za pomocą klas FileInfo i File	263
5.10. Podsumowanie	265
5.11. Sprawdź, czego się nauczyłeś	266

Rozdział 6. Budowanie aplikacji Windows Forms	269
6.1. Programowanie aplikacji Windows Forms	270
Ręczne konstruowanie aplikacji Windows Forms	271
6.2. Klasy kontrolki przestrzeni nazw Windows.Forms	274
Klasa Control	274
Praca z kontrolkami	276
Zdarzenia związane z kontrolkami	281
6.3. Klasa Form	286
Konfigurowanie wyglądu formularza	288
Ustawianie położenia i rozmiaru formularza	291
Wyświetlanie formularzy	292
Cykl życia formularza niemodalnego	293
Wzajemne oddziaływanie formularzy — przykład prostej aplikacji	295
Formularz właściciela i formularze własności	298
Okna komunikatów i okna dialogowe	300
Formularze MDI	302
6.4. Praca z menu	306
Właściwości klasy MenuItem	306
Menu kontekstowe	307
6.5. Dodawanie pomocy do formularza	308
Podpowiedzi	309
Obsługa zdarzeń naciśnięcia klawisza F1 i kliknięcia przycisku Pomoc	310
Komponent HelpProvider	312
6.6. Dziedziczenie formularzy	313
Konstruowanie i korzystanie z bibliotek formularzy	313
Korzystanie z formularzy potomnych	314
6.7. Podsumowanie	315
6.8. Sprawdź, czego się nauczyłeś	316
Rozdział 7. Kontrolki formularzy Windows Forms	317
7.1. Przegląd dostępnych w technologii .NET kontrolki formularzy Windows Forms	318
7.2. Klasy przycisków, grup kontrolki, paneli i etykiet	321
Klasa Button	321
Klasa CheckBox	323
Klasa RadioButton	324
Klasa GroupBox	325
Klasa Panel	326
Klasa Label	328
7.3. Kontrolki PictureBox i TextBox	329
Klasa PictureBox	329
Klasa TextBox	331
7.4. Klasy ListBox, CheckedListBox i ComboBox	333
Klasa ListBox	333
Pozostałe kontrolki list: ComboBox i CheckedListBox	338
7.5. Klasy ListView i TreeView	339
Klasa ListView	339
Klasa TreeView	346
7.6. Klasy ProgressBar, Timer i StatusStrip	351
Konstruowanie obiektów kontrolki StatusStrip	351
7.7. Konstruowanie kontrolki niestandardowych	353
Rozbudowa istniejącej kontrolki	354
Budowa niestandardowej kontrolki użytkownika	354

Przykład kontrolki UserControl	355
Stosowanie niestandardowych kontrolki użytkownika	356
Praca z kontrolkami użytkownika na etapie projektowania aplikacji	357
7.8. Stosowanie techniki przeciągania i upuszczania w kontrolkach formularzy WinForms	358
Wprowadzenie do techniki przeciągania i upuszczania	358
7.9. Korzystanie z zasobów	364
Praca z plikami zasobów	364
Stosowanie plików zasobów w procesie tworzenia formularzy obsługujących wiele języków	368
7.10. Podsumowanie	371
7.11. Sprawdź, czego się nauczyłeś	371
Rozdział 8. Elementy graficzne biblioteki GDI+ w technologii .NET	373
8.1. Przegląd biblioteki GDI+	374
Klasa Graphics	375
Zdarzenie Paint	379
8.2. Używanie obiektów klasy Graphics	382
Podstawowe operacje grafiki dwuwymiarowej	383
Pióra	387
Pędzle	389
Kolory	393
Przykładowy projekt: budowa przeglądarki kolorów	396
8.3. Obrazy	400
Wczytywanie i zapisywanie obrazów	400
Operacje na obrazach	403
Przykładowy projekt: praca z obrazami	406
Uwaga odnośnie biblioteki GDI i BitBit dla platformy Microsoft Windows	412
8.4. Podsumowanie	413
8.5. Sprawdź, czego się nauczyłeś	414
Rozdział 9. Czcionki, tekst i techniki drukowania	417
9.1. Czcionki	418
Rodziny czcionek	419
Klasa Font	420
9.2. Rysowanie łańcuchów tekstowych	424
Rysowanie tekstu wielowierszowego	424
Formatowanie łańcuchów za pomocą klasy StringFormat	425
Stosowanie pozycji tabulacyjnych	425
Przycinanie, wyrównywanie i zawijanie łańcuchów	428
9.3. Drukowanie	429
Wprowadzenie	430
Klasa PrintDocument	431
Ustawienia drukarki	431
Ustawienia strony	434
Zdarzenia PrintDocument	435
Zdarzenie PrintPage	437
Podgląd raportu przeznaczonego do wydrukowania	438
Przykład raportu	439
Tworzenie niestandardowych klas potomnych względem klasy PrintDocument	443
9.4. Podsumowanie	445
9.5. Sprawdź, czego się nauczyłeś	446

Rozdział 10. Praca z formatem XML w technologii .NET	447
10.1. Praca z formatem XML	448
Tworzenie dokumentów XML za pomocą mechanizmu serializacji do formatu XML ..	449
Definicja schematu XML (XSD)	452
Stosowanie arkuszy stylów XML	454
10.2. Techniki odczytu danych w formacie XML	457
Klasa XmlReader	458
Klasa XmlNodeReader	463
Klasa XmlReaderSettings	464
Weryfikacja danych XML z wykorzystaniem schematu XML	464
Opcje związane z odczytem danych XML	466
10.3. Techniki zapisywania danych w formacie XML	466
10.4. Przeszukiwanie kodu XML za pomocą instrukcji języka XPath	469
Konstruowanie zapytań języka XPath	470
Klasa XmlDocument i instrukcje języka XPath	473
Klasa XPathDocument i instrukcje języka XPath	473
Klasa XmlDataDocument i instrukcje języka XPath	474
10.5. Podsumowanie	476
10.6. Sprawdź, czego się nauczyłeś	477
Rozdział 11. ADO.NET	479
11.1. Architektura ADO.NET	480
Dostawca danych OLE DB w .NET	481
Dostawca danych .NET	481
11.2. Modele dostępu do danych — połączeniowy i bezpołączeniowy	483
Model połączeniowy	484
Model bezpołączeniowy	485
11.3. Model połączeniowy ADO.NET	487
Klasy do obsługi połączenia	487
Obiekt polecenia	490
Obiekt DataReader	494
11.4. Klasy DataSet, DataTable i model bezpołączeniowy	497
Klasa DataSet	497
Kolekcja DataTable	497
Ładowanie danych do obiektu DataSet	501
Używanie obiektu DataAdapter do aktualizacji bazy danych	503
Definiowanie relacji między tabelami w obiekcie DataSet	506
Wybór między modelem połączeniowym a bezpołączeniowym	509
11.5. XML i ADO.NET	509
Używanie obiektów DataSet do tworzenia plików z danymi i szablonami XML	510
Tworzenie szablonu obiektu DataSet na podstawie pliku XML	512
Wczytywanie danych XML do obiektu DataSet	512
11.6. Podsumowanie	515
11.7. Sprawdź, czego się nauczyłeś	515
Rozdział 12. Wiązanie danych z kontrolkami Windows Forms	517
12.1. Wprowadzenie do wiązania danych	518
Proste wiązanie danych	518
Złożone wiązanie danych z kontrolkami obsługującymi listy	520
Jednostronne i dwustronne wiązanie danych	521
Użycie menedżerów wiązania	523

12.2. Używanie prostego i złożonego wiązania danych w aplikacjach	526
Wiązanie z obiektem DataTable	526
Wiązanie kontroltek z obiektem ArrayList	528
Dodawanie elementów do źródła danych	530
Identyfikacja aktualizacji	531
Aktualizacja oryginalnej bazy danych	532
12.3. Klasa DataGridView	532
Właściwości	532
Zdarzenia	539
Ustawianie relacji nadrzędna-szczegółowa w kontrolkach DataGridView	543
Tryb wirtualny	545
12.4. Podsumowanie	550
12.5. Sprawdź, czego się nauczyłeś	550

Część III Zaawansowane techniki języka C# i platformy .NET

553

Rozdział 13. Programowanie asynchroniczne i wielowątkowość

555

13.1. Czym jest wątek?	556
Wielowątkowość	556
13.2. Programowanie asynchroniczne	559
Delegacje asynchroniczne	560
Przykłady implementacji wywołań asynchronicznych	563
13.3. Bezpośrednie używanie wątków	571
Tworzenie i używanie wątków	571
Praktyczne wykorzystanie wielowątkowości	574
Używanie puli wątków	577
Klasy Timer	578
13.4. Synchronizacja wątków	580
Atrybut Synchronization	582
Klasa Monitor	583
Muteksy	584
Semafor	585
Unikanie zakleszczenia	587
Podsumowanie technik synchronizacji	588
13.5. Podsumowanie	589
13.6. Sprawdź, czego się nauczyłeś	590

Rozdział 14. Tworzenie aplikacji rozproszonych za pomocą technologii Remoting

593

14.1. Domeny aplikacji	594
Zalety domen aplikacji	594
Domeny aplikacji i zestawy	595
Używanie klasy AppDomain	596
14.2. Zdalne korzystanie z obiektów	598
Architektura technologii Remoting	600
Rodzaje wywołań zdalnych	603
Obiekty aktywowane przez klienta	604
Obiekty aktywowane przez serwer	605
Rejestracja typów	606
Zdalne wywołania obiektów aktywowanych przez serwer	609
Zdalne wywoływanie obiektów aktywowanych przez klienta	616
Projektowanie aplikacji rozproszonych	621

14.3. Dzierżawy i sponsorowanie	622
Dzierżawa	623
Sponsorowanie	626
14.4. Podsumowanie	628
14.5. Sprawdź, czego się nauczyłeś	628

Rozdział 15. Usprawnianie kodu, bezpieczeństwo i instalacja 631

15.1. Stosowanie standardów programowania platformy .NET	633
Używanie FxCop	633
15.2. Zestawy z silną nazwą	636
Tworzenie zestawów z silną nazwą	637
Podpisywanie opóźnione	638
Globalna pamięć podręczna zestawów (GAC)	639
Kontrola wersji	640
15.3. Bezpieczeństwo	641
Uprawnienia i zbiory uprawnień	642
Dowód	646
Zasady bezpieczeństwa	648
Konfigurowanie zasad bezpieczeństwa	650
Narzędzie konfiguracyjne platformy .NET	651
Konfiguracja bezpieczeństwa opartego na uprawnieniach kodu za pomocą narzędzia konfiguracyjnego — przykład praktyczny	653
Żądanie uprawnień dla zestawu	657
Zabezpieczanie programowe	660
15.4. Instalowanie aplikacji	667
Instalacja w systemie Windows — XCOPY a instalator Windows	667
Instalowanie zestawów w globalnej pamięci podręcznej zestawów	668
Instalowanie zestawów prywatnych	668
Konfigurowanie za pomocą elementu codeBase	669
Używanie pliku konfiguracyjnego do zarządzania różnymi wersjami zestawów	670
Wersje zestawu i informacje o produkcie	670
15.5. Podsumowanie	671
15.6. Sprawdź, czego się nauczyłeś	672

Część IV Programowanie aplikacji internetowych

675

Rozdział 16. Formularze i kontrolki ASP.NET 677

16.1. Komunikacja klient-serwer poprzez połączenie internetowe	678
Przykładowa aplikacja internetowa	680
Kalkulator BMI w języku ASP.NET	684
Model wykorzystujący kod inline	685
Model wykorzystujący kod ukryty	691
Kod ukryty i klasy częściowe	695
Klasa Page	696
16.2. Kontrolki formularzy Web	699
Przegląd kontrolek Web	700
Określanie wyglądu kontrolek Web	701
Proste kontrolki	701
Kontrolki do obsługi list	706
Kontrolka DataList	708

16.3. Wiązanie danych oraz kontrolki do obsługi źródeł danych	711
Wiązanie z obiektem DataReader	711
Wiązanie z obiektem DataSet	712
Kontrolki do obsługi źródeł danych	714
16.4. Kontrolki walidacyjne	721
Używanie kontrolki walidacyjnych	721
16.5. Strony główne i strony z treścią	725
Tworzenie strony głównej	726
Tworzenie stron z treścią	727
Dostęp do strony głównej ze stron z treścią	728
16.6. Tworzenie i używanie niestandardowych kontrolki Web	729
Przykładowa niestandardowa kontrolka	729
Używanie niestandardowych kontrolki	732
Zarządzanie stanem kontrolki	732
Kontrolki złożone	733
16.7. Wybór kontrolki Web do wyświetlania danych	736
16.8. Podsumowanie	737
16.9. Sprawdź, czego się nauczyłeś	737
Rozdział 17. Środowisko aplikacji ASP.NET	739
17.1. Klasy HttpRequest i HttpResponse	740
Obiekt HttpRequest	741
Obiekt HttpResponse	744
17.2. ASP.NET i pliki konfiguracyjne	748
Zawartość pliku web.config	749
Dodawanie niestandardowych elementów konfiguracyjnych	753
17.3. Bezpieczeństwo aplikacji ASP.NET	756
Uwierzalnianie przy użyciu formularzy	757
Przykład zastosowania uwierzalniania przy użyciu formularzy	759
17.4. Przechowywanie stanu	763
Stan aplikacji	765
Stan sesji	766
17.5. Pamięć podręczna	769
Umieszczanie w pamięci podręcznej odpowiedzi	769
Umieszczanie danych w pamięci podręcznej	772
17.6. Tworzenie klienta używającego klas WebRequest i WebResponse	775
Klasy WebRequest i WebResponse	775
Przykładowy klient internetowy	775
17.7. Potoki HTTP	777
Przetwarzania żądania w potoku	778
Klasa HttpApplication	779
Moduły HTTP	782
Obiekty obsługi HTTP	787
17.8. Podsumowanie	790
17.9. Sprawdź, czego się nauczyłeś	791
Rozdział 18. Usługi Web	793
18.1. Wprowadzenie do usług Web	794
Wyszukiwanie i używanie usług Web	796
18.2. Tworzenie usługi Web	798
Samodzielne tworzenie usługi Web	799
Tworzenie usługi Web za pomocą środowiska VS.NET	802
Rozszerzanie usług Web za pomocą atrybutów WebService i WebMethod	803

18.3. Tworzenie klienta usługi Web	806
Tworzenie prostego klienta używającego klasy usługi Web	807
Tworzenie pośrednika za pomocą środowiska Visual Studio .NET	814
18.4. Język WSDL i protokół SOAP	815
Język opisu usług Web (WSDL)	815
Prosty protokół dostępu do obiektów (SOAP)	818
18.5. Używanie złożonych typów danych w usługach Web	825
Usługa Web zwracająca rysunki	825
Używanie usług Web Amazonu	827
Tworzenie pośrednika dla usług Web Amazonu	829
Tworzenie klienta usługi Web za pomocą formularzy Windows	830
18.6. Wydajność usług Web	832
Konfigurowanie połączenia HTTP	833
Obsługa dużych ilości danych	834
18.7. Podsumowanie	835
18.8. Sprawdź, czego się nauczyłeś	835

Dodatki **837**

Dodatek A Elementy wprowadzone w .NET 2.0 i C# 2.0	839
---	------------

Dodatek B Zdarzenia i delegacje kontrolki DataGridView	843
---	------------

Odpowiedzi do pytań	853
----------------------------------	------------

Skorowidz	869
------------------------	------------

1

Wprowadzenie do technologii .NET i języka C#

Zagadnienia omawiane w tym rozdziale:

- Przegląd składników platformy .NET — architektura i najważniejsze cechy technologii .NET.
- Wspólne środowisko uruchomieniowe (CLR) — przegląd zadań wykonywanych przez część uruchomieniową (wykonawczą) platformy .NET Framework, czyli funkcjonowania kompilatora „na bieżąco” (JIT), mechanizmów wczytywania zestawów oraz technik weryfikacji kodu.
- Wspólny system typów (CTS) i specyfikacja wspólnego języka (CLS) — mechanizmy wspólnego środowiska uruchomieniowego (CLR) w zakresie zapewniania zgodności i możliwości współpracy języków programowania.
- Zestawy .NET — analiza struktury zestawów, filozofii tych zestawów oraz różnic pomiędzy zestawami prywatnymi a zestawami współdzielonymi.
- Biblioteka klas platformy (FCL) — biblioteka platformy .NET oferuje setki klas bazowych pogrupowane w ramach logicznych przestrzeni nazw.
- Narzędzia programowania — wraz z platformą .NET otrzymujemy wiele narzędzi ułatwiających wytwarzanie kodu źródłowego. Mamy do dyspozycji między innymi narzędzie Ildasm do odwracania kompilacji kodu, WinCV do przeglądania właściwości klasy oraz narzędzie konfiguracyjne Framework Configuration.
- Kompilowanie i uruchamianie programów napisanych w języku C# — korzystanie z kompilatora języka programowania C# z poziomu wiersza poleceń wraz z omówieniem opcji decydujących o strukturze kompilowanej aplikacji.

Efektywne używanie języka programowania wymaga czegoś więcej niż dobrej znajomości jego składni i oferowanych mechanizmów. W praktyce podczas poznawania najnowszych technologii coraz większą rolę odgrywa dogłębne studiowanie samych środowisk programowania. Nie wystarczy więc opanowanie do perfekcji samego języka C# — wydajny

i skuteczny programista czy architekt oprogramowania musi też nabrać biegłości w korzystaniu z odpowiednich bibliotek klas i narzędzi umożliwiającej zagłębianie się w te biblioteki, diagnozowanie kodu oraz sprawdzanie faktycznej efektywności stosowanych konstrukcji programistycznych.

Celem tego rozdziału jest uświadomienie Czytelnikom istnienia najważniejszych rozwiązań oferowanych w ramach środowiska .NET, zanim jeszcze przystąpią do nauki składni i semantyki języka programowania C#. Skupimy się przede wszystkim na tym, jak to środowisko (nie język) wpływa na sposób wytwarzania oprogramowania. Jeśli nie masz doświadczenia w korzystaniu z technologii .NET, powinieneś uważnie przestudiować kilka nowych koncepcji, które tam wprowadzono. .NET zmienia sposób myślenia o dziedzicznym (często przestarzałym) kodzie oraz kontroli wersji, zmienia sposób dysponowania zasobami programowymi, umożliwia korzystanie z kodu napisanego w jednym z poziomów innego języka, upraszcza wdrażanie kodu, ponieważ eliminuje niepotrzebne związki z rejestrem systemowym, oraz tworzy samo-opisujący metajęzyk, który może być wykorzystywany do określania logiki programu w czasie jego wykonywania. Z każdym z tych elementów będziesz miał do czynienia na różnych etapach procesu wytwarzania oprogramowania i każdy z nich będzie miał wpływ na sposób projektowania i wdrażania Twoich aplikacji.

Z punktu widzenia programisty platforma .NET składa się ze środowiska uruchomieniowego (wykonawczego) połączonego z biblioteką klas bazowych. Organizacja tego rozdziału odpowiada właśnie takiemu postrzeganiu platformy .NET. Rozdział zawiera dwa osobne podrozdziały poświęcone odpowiednio wspólnemu środowisku uruchomieniowemu (ang. *Common Language Runtime* — *CLR*) oraz bibliotece klas platformy (ang. *Framework Class Library* — *FCL*). W dalszej części tego rozdziału zostaną przedstawione podstawowe narzędzia, dzięki którym programista może nie tylko w sposób wręcz intuicyjny realizować pewne zadania w procesie wytwarzania oprogramowania dla platformy .NET, ale też zarządzać i dystrybuować aplikacje. W ostatnim podrozdziale, swoistym preludium rozdziału 2., Czytelnik zostanie wprowadzony w świat kompilatora C# wraz z kilkoma przykładami praktycznych zastosowań.

1.1. Przegląd składników platformy .NET

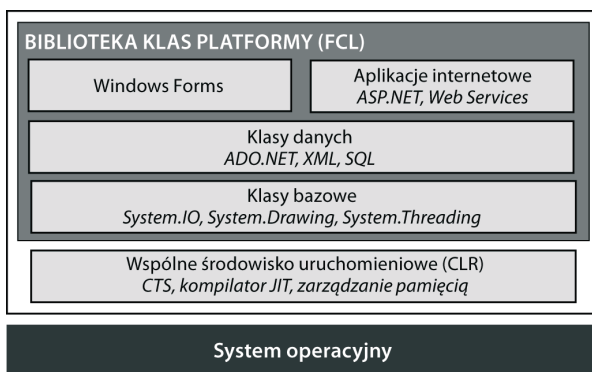
Platformę .NET zaprojektowano jako zintegrowane środowisko, które w założeniu ma umożliwiać bezproblemowe wytwarzanie i uruchamianie aplikacji internetowych, aplikacji bazujących na Windows Forms, a nawet aplikacji dla urządzeń mobilnych (z wykorzystaniem platformy Compact Framework). Na poniższej liście wymieniono podstawowe cele, jakie postawili sobie twórcy tej platformy:

- Zapewnienie logicznie spójnego, obiektowego środowiska dla rozmaitych aplikacji.
- Zapewnienie środowiska, które pozwoli zminimalizować konflikty wersji (popularnie nazywane „piekłem DLL”), które przez wiele lat prześladowały programistów Windows (COM) i — tym samym — uprościć proces dystrybucji (instalacji) kodu.

- Zapewnienie przenośnego środowiska, które będzie bazowało na certyfikowanych standardach i które będzie dawało możliwość zarządzania przez dowolny system operacyjny. Już teraz język programowania C# i podstawowy składnik środowiska uruchomieniowego technologii .NET, infrastruktura wspólnego języka (ang. *Common Language Infrastructure* — *CLI*), są objęte odpowiednimi standardami ECMA¹.
- Zapewnienie środowiska zarządzanego, w którym weryfikacja kodu pod kątem bezpieczeństwa wykonywania będzie możliwie prosta.

Aby zrealizować te ambitne cele, projektanci platformy .NET Framework opracowali dość specyficzną architekturę, która dzieli tę platformę na dwie części: wspólne środowisko uruchomieniowe (CLR) oraz bibliotekę klas platformy (FCL). Na rysunku 1.1 przedstawiono graficzną reprezentację zastosowanego podejścia.

Rysunek 1.1.
Platforma .NET



Środowisko uruchomieniowe CLR, czyli opracowana przez firmę Microsoft implementacja standardu CLI, obsługuje wykonywanie kodu i wszystkie związane z tym zadania, czyli kompilację, zarządzanie pamięcią, zapewnianie bezpieczeństwa, zarządzanie wątkami oraz wymuszanie bezpieczeństwa typów. O kodzie wykonywanym we wspólnym środowisku uruchomieniowym (CLR) mówimy, że jest kodem zarządzanym. Stosowanie tej terminologii pozwala odróżnić taki kod od kodu „niezarządzanego”, który nie implementuje wymagań związanych z funkcjonowaniem w środowisku CLR, czyli np. od obiektów COM czy komponentów bazujących na interfejsie programowym Windows API.

Inny ważny składnik technologii .NET, biblioteka klas platformy (FCL), jest — jak sama nazwa wskazuje — biblioteką kodu wielokrotnego użytku definiującą typy (klasy, struktury itp.), które są udostępniane aplikacjom wykonywanym w środowisku uruchomieniowym platformy .NET. Zgodnie z tym, co przedstawiono na rysunku 1.1, biblioteka FCL obejmuje klasy obsługujące dostęp do bazy danych, operacje graficzne, interakcję z kodem niezarządzanym, bezpieczeństwo oraz obsługę interfejsów użytkownika w postaci stron WWW i formularzy Windows Forms. Okazuje się, że z tej jednej biblioteki klas korzystają wszystkie języki przygotowane do współpracy z platformą .NET Framework. Oznacza to, że po zdobyciu doświadczenia w pracy ze wspomnianymi typami będziemy mogli stosować naszą wiedzę, pisząc kod w dowolnych językach programowania technologii .NET.

¹ W niniejszej książce we wszystkich odwołaniach do organizacji *ECMA International* (nazywanej dawniej *European Computer Manufacturers Association*) dla uproszczenia będziemy się posługiwali samym skrótem **ECMA**.

Microsoft .NET i standardy CLI

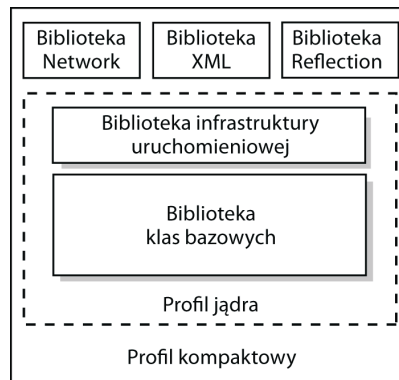
Naturalne pytanie stawiane przez programistów, którzy decydują się zainwestować swój czas w naukę języka programowania C# i mechanizmów platformy .NET, dotyczy tego, czy będą mogli wykorzystywać pozyskaną wiedzę także na innych platformach. W szczególności programiści chcą wiedzieć, czy technologia .NET, jako produkt Microsoftu, jest ściśle związana z systemem operacyjnym Windows czy może jest przenośną platformą wytwarzania i wdrażania oprogramowania, którą będzie można implementować dla konkurencyjnych systemów operacyjnych. Odpowiedź na to pytanie wymaga dobrego rozumienia relacji łączących technologię Microsoft .NET, język programowania C# oraz standardy infrastruktury wspólnego języka (CLI).

Standard infrastruktury wspólnego języka (CLI) definiuje niezależne od platformy środowisko wykonywania wirtualnego kodu. CLI nie preferuje żadnego systemu operacyjnego, zatem można go z powodzeniem stosować zarówno w systemie Linux, jak i w Windows. Centralnym elementem tego standardu jest definicja wspólnego języka pośredniego (ang. *Common Intermediate Language* — *CIL*), który musi być generowany przez kompilatory zgodne ze standardem CLI, oraz system typów, który definiuje typy danych obsługiwane przez wszystkie zgodne języki programowania. Język pośredni jest kompilowany do języka właściwego dla docelowego systemu operacyjnego (patrz następny podrozdział).

Infrastruktura wspólnego języka (CLI) obejmuje także standardy dla samego języka C#, który został stworzony i wypromowany przez firmę Microsoft, i jako taki jest de facto standardowym językiem programowania dla platformy .NET. Warto jednak pamiętać, że pozostali producenci szybko zainteresowali się standardem CLI i opracowali własne kompilatory dla takich języków jak Python, Pascal, Fortran, Cobol czy Eiffel .NET (przedstawiona lista oczywiście nie jest kompletna).

Platforma .NET (patrz rysunek 1.1) jest w istocie implementacją standardów CLI zaproponowaną przez firmę Microsoft. Najważniejszą cechą tej implementacji jest to, że oferuje daleko bardziej rozbudowany zestaw rozwiązań i funkcji niż ten zdefiniowany w standardowej infrastrukturze wspólnego języka. Aby się o tym przekonać, wystarczy porównać przedstawioną wcześniej strukturę platformy .NET z architekturą standardów CLI zaprezentowaną na rysunku 1.2.

Rysunek 1.2.
Architektura
zdefiniowana
przez specyfikację
infrastruktury
wspólnego
języka (CLI)



Najkrócej mówiąc, infrastruktura wspólnego języka (CLI) definiuje dwie implementacje: implementację minimalną, nazywaną **profilem jądra** (ang. *Kernel Profile*), oraz implementację bogatszą, nazywaną **profilem kompaktowym** (ang. *Compact Profile*). Profil jądra zawiera typy i klasy niezbędne do pracy kompilatorów zgodnych ze standardem CLI. Biblioteka klas bazowych (ang. *Base Class Library*) zawiera nie tylko podstawowe klasy typów danych, ale także klasy obsługujące proste operacje dostępu do plików, definiujące atrybuty zabezpieczeń oraz implementujące jednowymiarowe tablice. Profil kompaktowy wprowadza trzy dodatkowe biblioteki klas: bibliotekę XML definiującą proste mechanizmy analizy składniowej języka XML, bibliotekę Network obsługującą protokół HTTP i dostęp do portów oraz bibliotekę Reflection obsługującą **refleksje** (mechanizm analizy programu przez samego siebie za pośrednictwem metadanych).

Niniejsza publikacja byłaby dużo krótsza, gdyby zamiast implementacji zaproponowanej przez firmę Microsoft opisywała wyłącznie zalecenia standardu CLI. Moglibyśmy całkowicie zrezygnować z rozdziałów poświęconych bibliotece ADO.NET (zawierającej klasy obsługujące dostęp do baz danych), bibliotece ASP.NET (zawierającej klasy aplikacji internetowych) oraz bibliotece Windows Forms — co więcej, rozdziały poświęcone językowi XML byłyby nieporównanie krótsze. Jak się zapewne domyślasz, wymienione biblioteki są w jakimś stopniu uzależnione od funkcjonalności wykorzystywanego interfejsu Windows API. Twórcy technologii .NET dodatkowo przewidzieli możliwość wywoływania przez program interfejsu Win32 API za pośrednictwem mechanizmu nazwanego **Interop**. Oznacza to, że programista .NET ma dostęp nie tylko do interfejsu Win32 API, ale także do już działających aplikacji i komponentów (COM).

Ten dość szeroki most łączący technologię .NET z systemem Windows sprawia, że implementacja firmy Microsoft jest bardziej przezroczysta od środowisk wirtualnych — nie twierdę przy tym, że takie rozwiązanie jest nieuzasadnione. Dzięki temu programiści, którzy stawiają pierwsze kroki, tworząc oprogramowania dla platformy .NET, mogą — przynajmniej na początku — pisać aplikacje hybrydowe, czyli takie, które będą w sobie łączyły komponenty .NET z istniejącym kodem. Warto jednak pamiętać, że budowa takich hybryd wyklucza możliwość przenoszenia kodu gotowego do innych systemów operacyjnych.

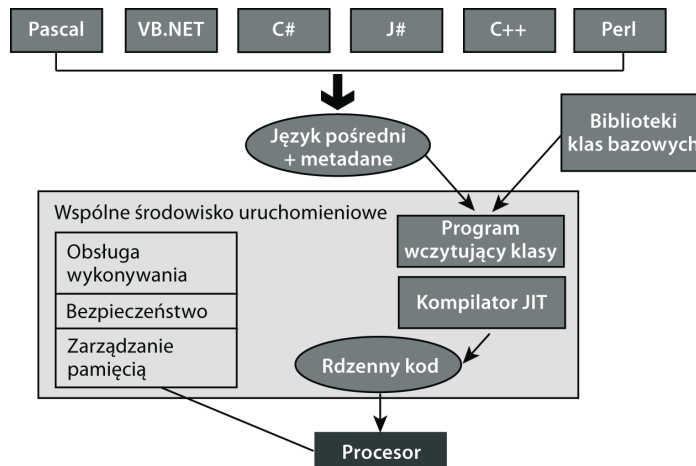
Mam też dobrą wiadomość dla programistów, a więc także Czytelników niniejszej książki — zaproponowane przez firmę Microsoft dodatkowe mechanizmy wzbudziły zainteresowanie szerszej społeczności, która zaangażowała się w opracowywanie implementacji CLI oferowanych z otwartym dostępem do kodu źródłowego. Zespołowi realizującemu jeden z czołowych projektów tego typu, nazwanego Mono², już teraz udało się zaimplementować większość mechanizmów bibliotek ADO.NET i Windows Forms, wszystkie klasy obsługujące język XML oraz bogaty zbiór klas obsługujących kolekcje. Jest to o tyle ważne, że wiedzę i doświadczenie zdobyte podczas pracy z technologią Microsoft .NET będzie można wykorzystać do implementowania podobnych rozwiązań dla takich platform jak Linux, BSD czy Solaris. Mając to na uwadze, spróbujmy teraz przeanalizować najważniejsze cechy implementacji standardu CLI zaproponowanej przez firmę Microsoft.

² Patrz strona internetowa http://www.mono-project.com/Main_Page.

1.2. Wspólne środowisko uruchomieniowe (CLR)

Wspólne środowisko uruchomieniowe (CLR) zarządza całym cyklem życia aplikacji: lokalizuje i kompiluje kod, wczytuje powiązane klasy, zarządza wykonywaniem aplikacji, a także zapewnia automatyczne zarządzanie pamięcią. Co więcej, środowisko CLR obsługuje całą integrację języków programowania, czyli de facto odpowiada za bezproblemową współpracę kodu zapisanego w różnych językach i skompilowanego za pomocą różnych kompilatorów. W niniejszym podrozdziale przyjrzymy się funkcjonowaniu wewnętrznych mechanizmów wspólnego środowiska uruchomieniowego, aby się przekonać, jak opisane przed chwilą cele faktycznie są realizowane. Nie będzie to co prawda analiza szczegółowa, ale z pewnością wystarczy do zapoznania się z terminologią, zaznajomienia z architekturą niezależną od języków programowania i zrozumienia, co rzeczywiście się dzieje w momencie, w którym tworzymy i uruchamiamy nasze programy.

Rysunek 1.3.
Funkcje wspólnego środowiska uruchomieniowego (CLR)



Kompilacja kodu platformy .NET

Kompilatory spełniające wymagania wspólnego środowiska uruchomieniowego (CLR) generują kod właśnie dla docelowego środowiska uruchomieniowego (wykonawczego), a nie dla konkretnego procesora. Generowany w ten sposób kod, nazywany wspólnym językiem pośrednim (CIL), językiem pośrednim (IL) lub językiem pośrednim Microsoft (MSIL), w praktyce ma postać kodu maszynowego umieszczanego w plikach EXE lub DLL. Warto pamiętać, że nie są to standardowe pliki wykonywalne, ponieważ dodatkowo wymagają użycia kompilatora **na bieżąco** (ang. *Just-in-Time*, w skrócie *JIT*), który w czasie wykonywania aplikacji dokona konwersji języka pośredniego na odpowiedni kod maszynowy. Ponieważ za zarządzanie tym językiem pośrednim odpowiada wspólne środowisko uruchomieniowe, kod zapisany w tym języku jest nazywany **kodem zarządzanym** (ang. *managed code*).

Kod pośredni jest kluczem do spełnienia formalnego celu stawianego platformie .NET, czyli zgodności różnych języków programowania. Zgodnie z tym, co przedstawiono na rysunku 1.3, wspólne środowisko uruchomieniowe (CLR) nie wie — bo nie musi wiedzieć — w którym

języku napisano daną aplikację. Środowisko CLR otrzymuje kod języka pośredniego (IL), który jest niezależny od stosowanych języków programowania. Ponieważ aplikacje komunikują się między sobą właśnie za pośrednictwem swojego kodu języka pośredniego, efekty pracy jednego kompilatora mogą być integrowane z kodem wygenerowanym przez inny kompilator.

Realizacja innego ważnego celu technologii .NET, przenośności, jest możliwa dzięki przeniesieniu procesu tworzenia kodu maszynowego na poziom kompilatora JIT. Oznacza to, że wygenerowany na jednej platformie kod języka pośredniego może być wykonywany na innej platformie zawierającej własną platformę i kompilator JIT, który generuje kod maszynowy właściwy dla tej platformy.

Poza generowaniem kodu języka pośredniego kompilatory spełniające wymagania wspólnego środowiska uruchomieniowego (CLR) dodatkowo muszą generować odpowiednie **metadane** w każdym module kodu. Metadane mają postać zbioru tabel, który zapewnia tym modułom kodu niezwykle ważną własność samoopisywania. Takie tabele zawierają informacje o **zestawie, pakiecie kodu** (ang. *assembly*), do którego należy dany kod, oraz kompletny opis samego kodu. Reprezentowane w ten sposób informacje obejmują między innymi dostępne typy, nazwy poszczególnych typów, składowe typów, atrybuty zasięgu (widoczności) typów oraz wszelkie inne cechy typów. Metadane mają wiele zastosowań:

- Najważniejszym użytkownikiem metadanych jest kompilator JIT, który wydobywa z nich wszystkie informacje o typach niezbędne do przeprowadzenia procesu kompilacji. Kompilator wykorzystuje te informacje także do weryfikacji kodu, aby mieć pewność, że dany program wykonuje swoje operacje prawidłowo. Przykładowo, kompilator JIT sprawdza prawidłowość wywołań metod, porównując wykorzystywane parametry wywołań z tymi, które zdefiniowano w metadanych odpowiednich metod.
- Metadane są wykorzystywane przez proces odpowiedzialny za **odzyskiwanie (czyszczenie) pamięci** (ang. *Garbage Collection*), czyli podstawowy składnik platformy .NET w dziedzinie zarządzania pamięcią. Proces GC (ang. *Garbage Collector*) używa metadanych do określania, kiedy pola w ramach danego obiektu odwołują się od innych pól i — tym samym — kiedy można, a kiedy nie można odzyskać pamięci zajmowanej przez te obiekty.
- Platforma .NET zawiera zbiór klas obsługujących funkcjonalność w zakresie odczytywania metadanych z wnętrza programu. Tego rodzaju funkcjonalność nazywa się zbiorczo **refleksją** (ang. *reflection*). Siłą tego rozwiązania jest możliwość wykonywania przez program zapytań odnośnie zawartości metadanych i podejmowania właściwych decyzji w zależności od uzyskiwanych odpowiedzi. Podczas lektury dalszej części tej książki przekonasz się, że właśnie refleksja jest kluczem do efektywnej pracy z niestandardowymi **atrybutami**, czyli obsługiwanymi w języku C# konstrukcjami w zakresie dodawania do programów niestandardowych metadanych (metadanych użytkownika).

Właśnie język pośredni (IL) i metadane decydują o możliwości wzajemnej współpracy wielu języków programowania, jednak faktyczny sukces tego rozwiązania zależy od tego, czy wszystkie kompilatory .NET rzeczywiście będą obsługiwały wspólny zbiór typów danych i specyfikacje języków. Przykładowo, dwa różne języki nie mogą być ze sobą w pełni zgodne

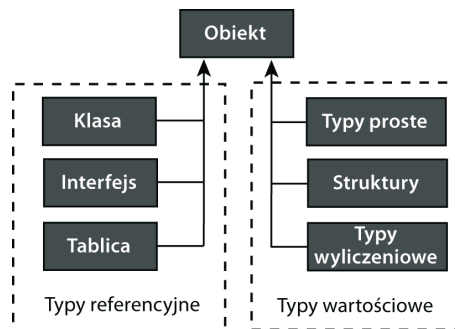
na poziomie kodu IL, jeśli jeden obsługuje 32-bitowe liczby całkowite ze znakiem, a drugi nie. Języki platformy .NET mogą się od siebie różnić składniowo (np. typ `int` w języku C# odpowiada typowi `Integer` w języku Visual Basic), ale musi między nimi istnieć zgodność bazowych typów reprezentowanych przez te często odmienne konstrukcje.

Jak już wspomniano, standard infrastruktury wspólnego języka (CLI) definiuje formalną specyfikację typów, nazwaną **wspólnym systemem typów** (ang. *Common Type System* — *CTS*), która jest integralną częścią wspólnego środowiska uruchomieniowego (CLR). System CTS definiuje nie tylko same typy, ale także zachowania gwarantujące pełną zgodność ze środowiskiem CLR.

Wspólny system typów (CTS)

Wspólny system typów (CTS) zawiera bazowy zbiór typów danych dla każdego języka zgodnego ze specyfikacją platformy .NET. CTS dodatkowo definiuje sposób deklarowania i tworzenia typów niestandardowych oraz mechanizm zarządzania cyklem życia instancji tych typów. Organizację wspólnego systemu typów (CTS) przedstawiono na rysunku 1.4.

Rysunek 1.4.
Typy bazowe zdefiniowane przez wspólny system typów (CTS)



Na podstawie powyższego rysunku można wywnioskować dwie rzeczy. Najbardziej oczywisty jest podział (kategoryzacja) typów na typy **referencyjne** (ang. *reference types*) i typy **wartościowe** (ang. *value types*). Ten podstawowy podział ma związek ze sposobem składowania typów i uzyskiwania dostępu do pamięci, w której są przechowywane — w dostępie do typów referencyjnych wykorzystuje się specjalny obszar pamięci, nazywany **stertą** (ang. *heap*), i pośrednictwo wskaźników; natomiast do typów wartościowych można się odwoływać bezpośrednio, ponieważ są składowane na stosie programu. Drugim ważnym wnioskiem płynącym z analizy rysunku 1.4 jest to, że wszystkie typy — zarówno typy niestandardowe, jak i typy definiowane przez platformę .NET — muszą dziedziczyć po jednym, predefiniowanym typie `System.Object`. Takie rozwiązanie daje nam pewność, że wszystkie typy obsługują podstawowy zbiór dziedziczonych po tym typie bazowym metod i właściwości.

W technologii .NET pojęcie „typu” jest bardzo ogólne — może się odnosić do klasy, struktury, typu wyliczeniowego, delegacji lub interfejsu.

Kompilator zgodny ze specyfikacją wspólnego systemu typów (CTS) musi gwarantować, że jego typy będą prawidłowo obsługiwane przez wspólne środowisko uruchomieniowe (CLR). Sama zgodność typu nie oznacza jeszcze, że dany język będzie się mógł komunikować z pozostałymi językami. Taka możliwość będzie wymagała zgodności z bardziej restrykcyjnym zbiorem **specyfikacji wspólnego języka** (ang. *Common Language Specification — CLS*), które definiują podstawowe reguły współpracy języków programowania. Wspomniane specyfikacje są w istocie minimalnym zestawem wymagań, które muszą być spełnione przez każdy kompilator zapewniający pełną zgodność ze środowiskiem CLR.

W tabeli 1.1 przedstawiono wybrany zbiór reguł specyfikacji wspólnego języka (CLS), które dobrze ilustrują, jakiego rodzaju własności należy uwzględniać podczas tworzenia typów zgodnych z tą specyfikacją (kompletną listę można znaleźć w dokumentacji zestawu narzędzi .NET SDK).

Tabela 1.1. Wybrane własności i reguły specyfikacji wspólnego języka (CLS)

Własności	Reguła
Widoczność (zasięg)	Reguły widoczności mają zastosowanie tylko dla tych składowych typów, które są udostępniane poza definiującym je zestawem .NET.
Znaki i wielkość liter	Aby dwie zmienne były traktowane jak osobne struktury, ich nazwy muszą się różnić czymś więcej niż tylko wielkością liter.
Typy podstawowe	Wymienione podstawowe typy danych są zgodne ze specyfikacją wspólnego języka (CLS): Byte, Int16, Int32, Int64, Single, Double, Boolean, Char, Decimal, IntPtr oraz String.
Wywołania konstruktorów	Konstruktor musi wywołać konstruktor klasy bazowej, zanim będzie mógł uzyskać dostęp do któregośkolwiek z jej pól.
Granice tablic	Dolna granica (najniższy indeks) wszystkich wymiarów tablic jest zero (0).
Typy wyliczeniowe	Typem bazowym wykorzystywanym w typach wyliczeniowych musi być Byte, Int16, Int32 lub Int64.
Sygnatury metod	Typy wszystkich parametrów i zwracanych wartości wykorzystywanych w sygnaturach metod muszą być zgodne ze specyfikacją wspólnego języka (CLS).

Przedstawione reguły są stosunkowo proste, ale też dość konkretne. Przyjrzyjmy się teraz fragmentowi kodu języka C#, aby lepiej zrozumieć, jak opisane metody są stosowane w praktyce.

```
public class Conversion
{
    public double Metric( double inches)
    { return (2.54 * inches); }
    public double metric( double miles)
    { return (miles / 0.62); }
}
```

Nawet jeśli nie masz doświadczenia w pracy z kodem języka C#, powinieneś bez trudu wskazać miejsce, w którym przedstawiony kod narusza reguły specyfikacji CLS. Zgodnie z drugą regułą opisaną w tabeli 1.1 nazwy konstrukcji językowych są uważane za różne, jeśli różnią się czymś więcej niż tylko wielkością znaków. Łatwo zauważyć, że nazwy metod

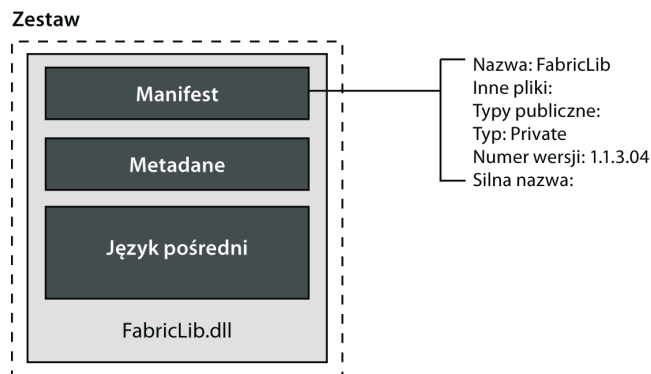
`Metric()` i `metric()` łamią tę regułę. Przedstawiony fragment kodu zostanie co prawda prawidłowo zinterpretowany przez kompilator języka C#, jednak już program napisany w języku Visual Basic.NET — który ignoruje wielkość liter — nie mógłby odróżnić nazw obu metod.

Zestawy .NET

Cały zarządzany kod wykonywany na platformie .NET musi się znajdować w specjalnych zestawach (nazywanych też pakietami kodu lub modułami .NET). Każdy taki zestaw logicznie jest przetwarzany jako pojedynczy plik EXE lub DLL. Fizycznie zestaw .NET może się jednak składać z całej kolekcji plików zawierających sam kod oraz niezbędne zasoby (np. obrazy lub dane w formacie XML).

Zestaw jest tworzony w chwili, gdy kompilator zgodny ze specyfikacją .NET konwertuje plik zawierający kod źródłowy na odpowiedni plik DLL lub EXE. Jak widać na rysunku 1.5, zestaw składa się z manifestu, metadanych oraz wygenerowanego przez kompilator języka pośredniego (IL). Poniżej przyjrzymy się poszczególnym składnikom zestawów nieco dokładniej.

Rysunek 1.5.
Zestaw .NET składający się z jednego pliku



Manifest. Każdy zestaw .NET musi zawierać pojedynczy plik zawierający tzw. **manifest**. Manifest jest zbiorem tabel metadanych, które zawierają listy nazw wszystkich plików w danym zestawie, referencje do zestawów zewnętrznych oraz informacje umożliwiające identyfikację danego zestawu (czyli np. nazwę i numer wersji). **Zestawy z silnymi nazwami** (ang. *strongly named assemblies* — patrz dalsza część tego rozdziału) dodatkowo zawierają unikatowe sygnatury cyfrowe. Podczas wczytywania zestawu .NET środowisko CLR w pierwszej kolejności otwiera właśnie plik zawierający manifest, aby mieć możliwość identyfikacji pozostałych składników danego zestawu.

Metadane. Poza opisanymi przed chwilą tabelami manifestu kompilator C# generuje jeszcze tabele definicji i referencji. Tabele definicji zawierają kompletny opis typów stosowanych w kodzie języka pośredniego. Przykładowo, kompilator C# generuje tabele definiujące typy, metody, pola, parametry i właściwości. Tabele referencji zawierają informacje o wszystkich referencjach do typów i pozostałych zestawów. Kompilator JIT wykorzystuje te tabele podczas konwertowania kodu języka pośredniego na rdzenny kod maszynowy.

Język pośredni. Rolę kodu języka pośredniego (IL) omówiliśmy we wcześniejszej części tego rozdziału. Zanim wspólne środowisko uruchomieniowe (CLR) będzie mogło użyć kodu tego języka, musimy ten kod umieścić w zestawach (plikach EXE lub DLL). Warto pamiętać, że wymienione rodzaje plików nie są identyczne — zestaw .NET w postaci pliku EXE musi definiować „punkt wejścia” (ang. *entry point*), od którego rozpocznie się jego wykonywanie, natomiast plik DLL pełni funkcję biblioteki kodu zawierającej definicje typów.

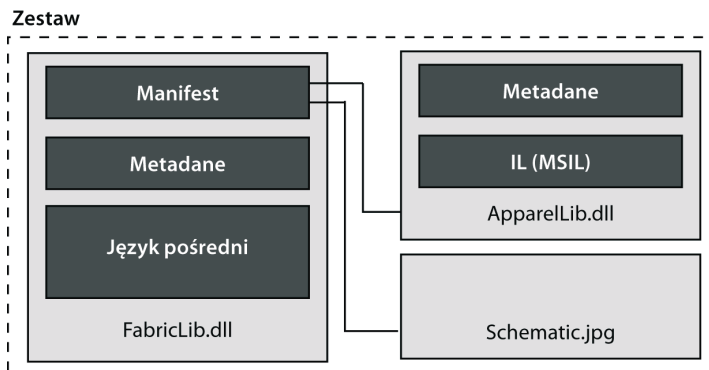
Zestaw jest czymś więcej niż tylko logicznym sposobem upakowywania kodu wykonywalnego. Zestawy są sercem zastosowanego w technologii .NET modelu rozmieszczania (wdrażania) kodu, kontroli wersji i zapewniania bezpieczeństwa:

- Cały zarządzany kod, niezależnie od tego, czy ma postać autonomicznego programu, kontrolki czy biblioteki DLL zawierającej typy wielokrotnego użytku, jest umieszczany właśnie w zestawach. Zestaw .NET jest jednostką najbardziej atomową, którą można wdrożyć w systemie. Do uruchomienia aplikacji wymagane są tylko te zestawy, bez których nie można przeprowadzić fazy inicjalizacji. Pozostałe zestawy są wczytywane na żądanie (według potrzeb). Doświadczeni programiści potrafią efektywnie wykorzystywać taki podział aplikacji — dzielić kod pomiędzy zestawy .NET w zależności od częstotliwości jego wykorzystywania.
- W żargonie programistów technologii .NET zestaw wyznacza tzw. **granicę wersji** (ang. *version boundary*). Pole wersji w manifeście jest stosowane dla wszystkich typów i zasobów w danym zestawie. Oznacza to, że wszystkie pliki składające się na ten zestaw są traktowane jak ta sama jednostka, oznaczona tą samą wersją. Dzięki takiemu oddzieleniu warstwy fizycznych zestawów od logicznego podziału w technologii .NET istnieje możliwość współużytkowania logicznych atrybutów przez wiele plików fizycznych. Ta wyjątkowa cecha odróżnia pliki zestawów od tradycyjnych, systemowych plików DLL.
- Zestaw wyznacza też **granicę bezpieczeństwa** (ang. *security boundary*), od której zależą wszelkie uprawnienia dostępu. W języku C# stosuje się tzw. **modyfikatory dostępu** (ang. *access modifiers*), które kontrolują dostęp do typów i składowych typów w ramach zestawów. Dwa takie modyfikatory są stosowane do wyznaczania granic bezpieczeństwa w dostępie do zestawów: modyfikator `public` zezwala na nieograniczony dostęp do zestawu .NET, natomiast modyfikator `internal` ogranicza ten dostęp do typów i składowych należących do tego samego zestawu.

Jak już wspomniano, zestaw może się składać z wielu plików. Nie muszą to być wyłącznie moduły kodu — w skład zestawu mogą wchodzić także pliki zasobów, w tym obrazy graficzne i pliki tekstowe. W typowych zastosowaniach tego rodzaju pliki mają na celu umożliwienie aplikacji wykorzystywanie zasobów niezbędnych do prezentacji interfejsu użytkownika z uwzględnieniem ustawień kraju i języka docelowego użytkownika. Warto pamiętać, że nie ma żadnych ograniczeń co do liczby plików w zestawie. Na rysunku 1.6 przedstawiono strukturę wieloplikowego zestawu .NET.

Przedstawiony diagram wieloplikowego zestawu pokazuje wyraźnie, że manifest zestawu zawiera informacje identyfikujące wszystkie pliki wchodzące w skład tego zestawu.

Rysunek 1.6.
Zestaw
wieloplikowy



Chociaż większość zestawów składa się z pojedynczego pliku, istnieje przynajmniej kilka przypadków, w których lepszym rozwiązaniem jest stosowanie zestawów wieloplikowych:

- Zestawy wieloplikowe umożliwiają łączenie modułów przygotowanych w różnych językach programowania. Programiści zaangażowani w prace nad oprogramowaniem mogą korzystać z języka Visual Basic.NET do błyskawicznego wytwarzania aplikacji (ang. *Rapid Application Development* — *RAD*) oraz z języka C# do budowy komponentów lub rozwiązań korporacyjnych. Kod stworzony w obu językach może nie tylko współistnieć, ale też efektywnie współpracować w ramach jednego zestawu .NET.
- Moduły kodu można podzielić w sposób pozwalający zoptymalizować proces wczytywania kodu przez wspólne środowisko uruchomieniowe (CLR). Wzajemnie powiązany i często wykorzystywany kod powinien się znaleźć w jednym module, natomiast kod, z którego korzystamy stosunkowo rzadko, powinien trafić do innego modułu. Środowisko CLR nigdy nie wczytuje modułów do momentu, w którym rzeczywiście będą potrzebne. Tworząc bibliotekę klas, powinieneś iść krok dalej i pogrupować komponenty według cyklu życia, wersji i zabezpieczeń (być może różnice w tych obszarach będą uzasadniały podział klas pomiędzy wiele zestawów).
- Pliki zasobów mogą tworzyć własny moduł, który będzie odseparowany od modułów języka pośredniego (IL). Takie rozwiązanie ułatwi korzystanie ze wspólnych zasobów przez wiele aplikacji.

Wieloplikowe zestawy można tworzyć albo za pomocą kompilatora C# uruchamianego z poziomu wiersza poleceń, albo za pomocą narzędzia łączącego zestawy .NET, czyli Assembly Linkera, *Al.exe*. W ostatnim podrozdziale tego rozdziału przedstawiono przykład użycia do tego celu kompilatora języka C#. Warto pamiętać, że w środowisku Visual Studio.NET 2005 tworzenie wieloplikowych zestawów nie jest obsługiwane.








Zestawy prywatne i współdzielone

Zestawy mogą być wdrażane na dwa sposoby: jako struktury prywatne lub struktury globalne. Zestawy znajdujące się w katalogu bazowym aplikacji lub w jego podkatalogach są nazywane **zestawami wdrażanymi prywatnie** lub po prostu **zestawami prywatnymi** (ang. *privately deployed assemblies*). Instalacja i aktualizacja zestawów prywatnych nie mogłaby

być łatwiejsza — zwykle wymaga tylko skopiowania zestawu do katalogu bazowego, nazywanego *AppBase*, w którym znajduje się dana aplikacja. Nie są wymagane żadne zmiany w ustawieniach rejestru. Co więcej, istnieje możliwość dodania pliku konfiguracyjnego aplikacji, który przykryje ustawienia zapisane w manifeście i umożliwi przenoszenie plików zestawów w ramach katalogu *AppBase*.

Zestawy współdzielone to takie, które są instalowane w lokalizacji globalnej, nazywanej **globalną pamięcią podręczną zestawów** (ang. *Global Assembly Cache* — *GAC*), skąd są udostępniane wielu aplikacjom. Najważniejszą zaletą globalnej pamięci podręcznej zestawów jest możliwość równoczesnego korzystania z wielu wersji tych samych zestawów. Właściwa realizacja tej koncepcji wymagała od twórców technologii .NET wyeliminowania problemu konfliktów nazw, który jest prawdziwą plagą w tradycyjnych bibliotekach DLL — zdecydowano się na użycie aż czterech różnych atrybutów identyfikujących zestawy: nazwy pliku, ustawień kulturowych (regionalnych), numeru wersji oraz tokenu klucza publicznego.

Publiczne zestawy zwykle są składowane w katalogu *assembly* zlokalizowanym gdzieś wewnątrz katalogu systemu operacyjnego (czyli np. wewnątrz katalogu *WINNT* w przypadku systemu Microsoft Windows 2000). Na rysunku 1.7 przedstawiono zrzut ekranu, który dobrze ilustruje specjalny format wyświetlania wszystkich czterech atrybutów (platforma .NET zawiera plik DLL, który rozszerza funkcjonalność Eksploratora Windows o mechanizmy prezentacji zawartości pamięci podręcznej *GAC*). Przyjrzyjmy się teraz każdemu z tych czterech atrybutów.

Nazwa zestawu	Wersja	Ustawienia regionalne	Token klucza publicznego
 Accessibility	1.0.5000.0	neutral	b03f5f7f11d50a3a
 ADODB	7.0.3300.0	neutral	b03f5f7f11d50a3a
 CrystalDecisions.Enterprise.Desktop.Report	10.2.3600.0	neutral	692fba5521e1304
 CrystalDecisions.Enterprise.Framework	10.2.3600.0	neutral	692fba5521e1304
 CrystalDecisions.Enterprise.InfoStore	10.2.3600.0	neutral	692fba5521e1304
 CrystalDecisions.Enterprise.PluginManager	10.2.3600.0	neutral	692fba5521e1304
 CrystalDecisions.Enterprise.Viewing.ReportSource	10.2.3600.0	neutral	692fba5521e1304

Rysunek 1.7. Fragment listy zestawów wchodzących w skład globalnego katalogu zestawów

Nazwa zestawu. Nazywana także **przyjazną nazwą** (ang. *friendly name*), ma postać nazwy pliku bez rozszerzenia.

Wersja. Każdy zestaw ma przypisany numer wersji, który jest stosowany dla wszystkich plików wchodzących w skład tego zestawu. Numer wersji składa się z czterech liczb w następującym formacie:

<numer główny>.<numer dodatkowy>.<kompilacja>.<poprawka>

Główne i dodatkowe numery wersji zwykle są aktualizowane w przypadku wprowadzenia zmian, które zrywają zgodność wstecz (z wcześniejszymi wersjami). Numery wersji można przypisywać do zestawów, dołączając w ich kodzie źródłowym atrybut `AssemblyVersion`.

Ustawienia regionalne. Zawartość zestawu można przypisać do określonej kultury lub języka. Istnieje specjalny, dwuliterowy kod, w którym "en" oznacza język angielski, natomiast "fr" wskazuje na język francuski — tego rodzaju kody można przypisywać do zestawów, umieszczając w kodzie źródłowym atrybut `AssemblyCulture`:

```
[assembly: AssemblyCulture ("fr-CA")]
```

Token klucza publicznego. Aby zapewnić unikatowość i autentyczność współdzielonych zestawów, środowisko .NET wymaga od ich twórców oznaczania zestawów za pomocą tzw. **silnych nazw** (ang. *strong names*). Proces przypisywania takich nazw, nazywany **podpisywaniem** zestawów (ang. *signing*), wymaga stosowania par kluczy publicznych i prywatnych. Kiedy kompilator buduje dany zestaw, wykorzystuje klucz prywatny do wygenerowania silnej nazwy. Klucz publiczny jest na tyle duży, że konieczne jest utworzenie specjalnego tokenu (reprezentacji symbolicznej) przez zakodowanie za pomocą funkcji mieszającej całego klucza i wybraniu tylko ostatnich ośmiu bajtów. Tak utworzony token jest następnie umieszczany w manifeście każdego zestawu klienckiego, który odwołuje się do odpowiedniego zestawu współdzielonego, i jest wykorzystywany do identyfikacji tego zestawu w czasie wykonywania.

Zestaw .NET, który oznaczono (podpisano) za pomocą kluczy publicznego i prywatnego, jest określany mianem zestawu z silną nazwą lub silnie nazwanego zestawu. Wszystkie zestawy współdzielone muszą mieć przypisane silne nazwy.

Prekompilacja zestawu

Po wczytaniu zestawu należy skompilować kod języka pośredniego do postaci rdzennego kodu maszynowego. Jeśli masz już doświadczenie w pracy z plikami wykonywalnymi w formacie kodu maszynowego, zapewne będzie Cię interesowała wydajność tego rozwiązania i ewentualnie możliwość tworzenia odpowiedników „plików wykonywalnych” w środowisku .NET. Odpowiedź na drugie pytanie brzmi: „tak” — technologia .NET przewiduje możliwość prekompilowania zestawów.

Platforma .NET Framework zawiera narzędzie **generatora rdzennych obrazów** (ang. *Native Image Generator, Ngen*), które umożliwia kompilowanie zestawów do postaci „rdzennych obrazów” składanych w specjalnej pamięci podręcznej — zarezerwowanym obszarze wspomnianej już pamięci podręcznej GAC. Za każdym razem, gdy wspólne środowisko uruchomieniowe (CLR) wczytuje zestaw, przeszukuje tę pamięć podręczną pod kątem ewentualnej dostępności rdzennego obrazu; jeśli znajdzie taki obraz, wczytuje właśnie prekompilowany kod. Na pierwszy rzut oka wydaje się, że takie rozwiązanie jest korzystne pod względem wydajności. W rzeczywistości opisana strategia ma jednak wiele poważnych wad.

Narzędzie Ngen tworzy obraz dla hipotetycznej architektury sprzętowej, dzięki czemu będzie go można uruchomić np. na dowolnym komputerze z procesorem x86. Zupełnie inaczej działa kompilator JIT w środowisku .NET, który dysponuje niezbędną wiedzą o komputerze docelowym kompilacji i może odpowiednio dostosować podejmowane działania

optymalizacyjne. Efekt jest taki, że wersje generowane przez ten kompilator często przewyższają pod względem wydajności prekompilowane zestawy. Kolejną wadą rdzennych obrazów jest ryzyko dezaktualizacji prekompilowanych zestawów wskutek zmian w konfiguracji sprzętowej lub w systemie operacyjnym, np. w wyniku zainstalowania aktualizacji.

Dynamicznie kompilowane zestawy z reguły są pod względem wydajności lepsze lub równe prekompilowanym plikom wykonywalnym wygenerowanym przez narzędzie Ngen.

Weryfikacja kodu

W ramach procesu kompilacji JIT wspólne środowisko uruchomieniowe (CLR) przeprowadza dwa rodzaje **weryfikacji**: weryfikację kodu języka pośredniego oraz test poprawności metadanych. Celem tych działań jest sprawdzenie, czy odpowiedni kod faktycznie gwarantuje **bezpieczeństwo typów** (ang. *type safety*). W praktyce chodzi o upewnienie się, że typy parametrów w metodzie wywołującej i metodzie wywoływanej są ze sobą zgodne oraz że wartości zwracane przez daną metodę są zawsze zgodne z typem zadeklarowanym w deklaracji tej metody. Krótko mówiąc, wspólne środowisko uruchomieniowe (CLR) przeszukuje kod języka pośredniego (IL) pod kątem zgodności wartości przypisywanych do zmiennych z typami tych zmiennych; jeśli zostanie wykryta niezgodność w tym zakresie, kompilacja JIT zakończy się wygenerowaniem stosownego wyjątku.

Kod generowany przez kompilator C# domyślnie zapewnia bezpieczeństwo typów. Istnieje jednak słowo kluczowe `unsafe`, za pomocą którego możemy na poziomie programu napisanego w tym języku osłabić restrykcje w zakresie dostępu do danych (włącznie z dopuszczeniem możliwości stosowania odwołań wykraczających poza granice tablic).

Zaletą takiej weryfikacji jest pewność, że wykonywanie danego kodu nie będzie miało negatywnego wpływu na funkcjonowanie pozostałych aplikacji wskutek uzyskiwania dostępu do pamięci spoza przydzielonego obszaru. Oznacza to, że środowisko CLR może bezpiecznie uruchamiać wiele aplikacji w ramach jednego procesu lub przestrzeni adresowej, co z kolei przekłada się na lepszą wydajność i niższy poziom wykorzystania zasobów systemu operacyjnego.

1.3. Biblioteka klas platformy (FCL)

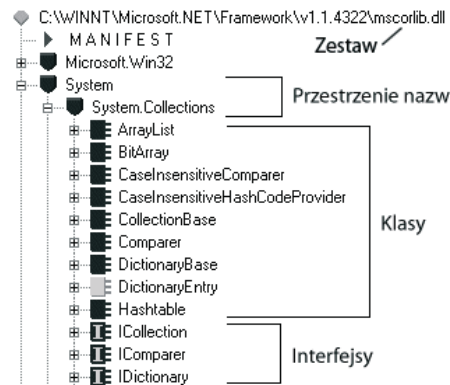
Biblioteka klas platformy (FCL) jest zbiorem klas i innych typów (typów wyliczeniowych, struktur i interfejsów), które są dostępne dla zarządzanego kodu napisanego w dowolnym języku zgodnym ze specyfikacją wspólnego środowiska uruchomieniowego (CLR). Jest to o tyle ważne, że eliminuje znaną z innych technologii konieczność wiązania bibliotek z konkretnymi kompilatorami. Programista, który raz zapozna się z typami udostępnianymi przez tę bibliotekę, będzie mógł tę wiedzę wykorzystywać podczas pracy z dowolnym wybranym przez siebie językiem programowania zgodnym z technologią .NET.

Zasoby w ramach biblioteki FCL podzielono pomiędzy logiczne grupy nazwane **przestrzeniami nazw** (ang. *namespaces*). W większości przypadków zakres funkcjonalności tych grup jest dość szeroki. Przykładowo typy wykorzystywane w operacjach graficznych pogrupowano w ramach przestrzeni nazw `System.Drawing` i `System.Drawing.Drawing2D`, natomiast typy niezbędne do wykonywania operacji wejścia-wyjścia na plikach umieszczono w przestrzeni nazw `System.IO`. Warto pamiętać, że przestrzenie nazw reprezentują logiczny podział typów, a nie fizyczny.

Biblioteka FCL składa się z setek zestawów (w postaci plików DLL), z których każdy może zawierać wiele przestrzeni nazw. Co więcej, pojedyncza przestrzeń nazw może obejmować wiele zestawów. Aby lepiej zrozumieć strukturę biblioteki FCL, przeanalizujmy przykładowy zestaw tej biblioteki (patrz rysunek 1.8).

Rysunek 1.8.

Fragment danych wynikowych wyświetlonych przez program Ildasm — lista przestrzeni nazw i typów składających się na dany zestaw



Na rysunku 1.8 przedstawiono fragment danych wyjściowych wygenerowanych przez narzędzie `Ildasm.exe` — na ich podstawie możemy przeanalizować zawartość zestawu `mscorlib`. Chociaż zaprezentowana lista obejmuje tylko część składników tego zestawu, możemy się przekonać, że zestaw `mscorlib` zawiera niezwykle ważną przestrzeń nazw `System`, która pełni funkcję repozytorium dla typów decydujących o podstawowej funkcjonalności technologii .NET. Zestaw `mscorlib` zawiera też przestrzeń nazw `System.Collection`, która — jak sama nazwa wskazuje — obejmuje klasy i interfejsy wykorzystywane podczas przetwarzania kolekcji danych.

W tabeli 1.2 wymieniono kilkanaście najważniejszych przestrzeni nazw biblioteki FCL platformy .NET. W ostatniej kolumnie zawarto numery rozdziałów tej książki, w których znajdziesz przykłady zastosowań odpowiednich przestrzeni nazw.

Przestrzenie nazw stanowią swoistą mapę drogową dla programistów przeglądających bibliotekę FCL w poszukiwaniu niezbędnych klas i interfejsów. Przykładowo, jeśli budowana aplikacja ma wykorzystywać interfejs oparty na stronach WWW, zapewne najbardziej będą nas interesowały typy przestrzeni nazw `System.Web.*`. Kiedy nauczysz się podstaw technologii .NET i zdobędziesz doświadczenie w pracy z językiem programowania C#, odkryjesz, że największym wyzwaniem jest właśnie zapoznanie się z wbudowanymi typami zawartymi w bibliotece klas platformy (FCL).

Tabela 1.2. Wybrane przestrzenie nazw biblioteki FCL

Przeźrezeń nazw	Zastosowania	Rozdział
System	Zawiera podstawowe typy danych wykorzystywane przez wszystkie aplikacje. Przestrzeń nazw System zawiera też klasy wyjątków, predefiniowane atrybuty, bibliotekę Math oraz klasy odpowiedzialne za zarządzanie środowiskiem aplikacji.	3., 18.
System.Collections System.Collections.Specialized System.Collections.Generic	Zawiera interfejsy i klasy wykorzystywane do zarządzania kolekcjami obiektów. Do kolekcji obsługiwanych przez tę przestrzeń nazw należą ArrayList, Hashtable i Stack.	4.
System.Data System.Data.OracleClient System.Data.SqlClient System.Data.OleDb System.Data.Odbc	Zawiera klasy wykorzystywane podczas wykonywania operacji na bazach danych (ADO.NET). Za obsługę systemów zarządzania bazami danych Oracle i SQL Server odpowiadają klienckie przestrzenie nazw, odpowiednio OracleClient i SqlConnection, natomiast przestrzenie OleDb i Odbc definiują operacje na połączeniach z bazami danych.	11., 12.
System.Diagnostics	Zawiera klasy, które mogą być wykorzystywane do śledzenia wykonywania programu, diagnozowania oraz przetwarzania systemowych dzienników zdarzeń i liczników wydajności.	13.
System.Drawing System.Drawing.Drawing2D System.Drawing.Printing System.Drawing.Text	Oferuje funkcjonalność graficzną dla interfejsu programowego GDI+. Wymienione przestrzenie nazw zawierają klasy obsługujące zarówno rysowanie, jak i korzystanie z rozmaitych piór, pędzli, kształtów geometrycznych i czcionek.	8., 9.
System.Globalization	Zawiera klasy definiujące informacje związane z ustawieniami regionalnymi, czyli sposobem reprezentowania dat, kwot pieniężnych i symbolami walut.	5.
System.IO	Definiuje operacje wejścia-wyjścia na plikach i strumieniach danych. Za pomocą tych klas można uzyskać dostęp do systemu plików obsługiwanego przez wykorzystywany system operacyjny.	5.
System.Net	Zawiera klasy obsługujące protokoły sieciowe i operacje w środowiskach sieciowych. Przykładowo, przestrzeń nazw System.Net definiuje klasy WebRequest i WebResponse, które reprezentują odpowiednio żądanie i odpowiedź przy pobraniu strony WWW.	17.
System.Reflection System.Reflection.Emit	Zawiera typy, dzięki którym można w czasie wykonywania aplikacji przetwarzać jej metadane. Przestrzeń nazw Emit umożliwia dynamiczne generowanie metadanych i kodu języka pośredniego (IL) za pomocą kompilatora lub innego narzędzia.	7., 15., dodatek B
System.Runtime.InteropServices	Zapewnia możliwość współpracy pomiędzy zarządzanym i niezarządzanym kodem, w tym tradycyjnymi bibliotekami DLL i obiektami COM.	8.

Tabela 1.2. Wybrane przestrzenie nazw biblioteki FCL (ciąg dalszy)

Przeźrzeń nazw	Zastosowania	Rozdział
System.Security System.Security.Permissions System.Security.Cryptography	Zawiera klasy odpowiedzialne za zarządzanie bezpieczeństwem w środowisku .NET. Definiuje klasy kontrolujące dostęp do operacji i zasobów.	5., 15.
System.Text.RegularExpressions	Zawiera klasy obsługujące wchodzący w skład środowiska .NET moduł przetwarzania wyrażeń regularnych.	5.
System.Threading System.Threading.Thread	Zarządza całym cyklem życia wątków — tworzeniem wątków, synchronizacją pracy wątków oraz dostępem do puli wątków.	13.
System.Web System.Web.Services System.Web.UI System.Web.UI.WebControls System.Web.Security	Zawiera klasy wykorzystywane do rozmaitych działań w ramach aplikacji internetowych (oferowane klasy łącznie są nazywane technologią ASP.NET). Klasy przestrzeni nazw System.Web zarządzają komunikacją na linii przeglądarka-serwer, przetwarzają znaczniki kontekstu użytkownika oraz zawierają kontrolki, które poprawiają wygląd i funkcjonalność stron internetowych. Przestrzeń nazw Web.Services zawiera klasy wymagane do przesyłania komunikatów XML za pośrednictwem protokołu SOAP. Przestrzeń nazw Web.UI zawiera klasy i interfejsy wykorzystywane do tworzenia kontrolki i stron internetowych składających się na formularze WWW.	16., 17., 18.
System.Windows.Forms	Zawiera klasy wykorzystywane do budowy graficznego interfejsu użytkownika dla aplikacji systemu Windows. Przestrzeń nazw Forms oferuje takie kontrolki jak ListBox, TextBox czy DataGridView.	6., 7.
System.Xml	Typy niezbędne do przetwarzania danych w formacie XML.	10.

1.4. Praca z platformą .NET i zestawem narzędzi .NET Framework SDK

Zestaw narzędzi .NET Framework *Software Development Kit* (SDK) zawiera narzędzia, kompilatory i dokumentację niezbędną do tworzenia oprogramowania, które będzie prawidłowo działało na każdym komputerze z zainstalowaną platformą .NET Framework. Wersję instalacyjną dla systemów Windows XP, Windows 2000, Windows Server 2003 i wszystkich nowszych wersji systemu operacyjnego Windows można pobrać za darmo z witryny internetowej firmy Microsoft (zajmuje około 100 megabajtów). Jeśli masz już zainstalowane środowisko Visual Studio .NET, nie będziesz musiał pobierać tego zestawu narzędzi, ponieważ VS .NET instaluje je automatycznie.

Użytkownicy oprogramowania zbudowanego z wykorzystaniem zestawu narzędzi .NET Framework SDK nie muszą tego zestawu instalować na swoich komputerach, powinni natomiast dysponować odpowiednią (zgodną) wersją platformy .NET Framework. Na witrynie internetowej firmy Microsoft³ udostępniono darmowy pakiet *.NET Framework Redistributable* (zajmuje ponad 20 megabajtów) — należy go zainstalować na wszystkich komputerach, których użytkownicy będą korzystali z aplikacji opracowanych w technologii .NET. Poza systemami wymienionymi przy okazji omawiania zestawu narzędzi SDK pakiet .NET Framework Redistributable można instalować także w systemach Windows 98 i Windows ME. Aplikacje .NET będą — z drobnymi wyjątkami — działały identycznie we wszystkich platformach systemów operacyjnych, ponieważ ich docelowym otoczeniem jest wspólne środowisko uruchomieniowe (CLR), a nie system operacyjny. Przed zainstalowaniem pakietu .NET Framework Redistributable powinniśmy się upewnić, że nasz system spełnia wymagania opisane na witrynie internetowej firmy Microsoft i dotyczące między innymi zainstalowanej przeglądarki internetowej Internet Explorer 5.01 lub nowszej.

Aktualizowanie platformy .NET Framework

W przeciwieństwie do wielu innych środowisk wytwarzania oprogramowania instalowanie nowej wersji platformy .NET nie stanowi żadnego problemu i nie wymaga od użytkownika większego wysiłku. Proces instalacji umieszcza zaktualizowaną wersję w nowym katalogu, którego nazwa odpowiada danej wersji platformy. Co ważne, nie występują żadne zależności pomiędzy plikami nowszych i starszych wersji tej platformy. Oznacza to, że wszystkie wersje zainstalowane w naszym systemie są w pełni funkcjonalne. Najczęściej poszczególne wersje platformy .NET Framework są składowane w następujących katalogach (ścieżki mogą się oczywiście różnić w zależności od systemu i zainstalowanych wersji):

```
\\winnt\Microsoft.NET\Framework\v1.0.3705
```

```
\\winnt\Microsoft.NET\Framework\v1.1.4322
```

```
\\winnt\Microsoft.NET\Framework\v2.0.40607
```

Każda instalacja nowej wersji oprogramowania platformy rodzi pytania o jej zgodność z aplikacjami opracowanymi z wykorzystaniem starszych wersji. Okazuje się, że twórcy technologii .NET znaleźli rozwiązanie ułatwiające uruchamianie istniejących aplikacji na dowolnej wersji platformy .NET. Kluczowym rozwiązaniem jest **plik konfiguracyjny aplikacji** (ang. *application configuration file* — szczegółowe omówienie tego rodzaju plików znajdziesz w rozdziale 15.). Jest to plik tekstowy zawierający znaczniki i elementy w formacie XML, które instruują wspólne środowisko uruchomieniowe (CLR) o właściwym sposobie wykonywania danej aplikacji. Plik konfiguracyjny aplikacji może określać położenie zewnętrznych zestawów, wersje tych zestawów oraz wersje platformy .NET Framework obsługiwane przez poszczególne aplikacje i (lub) komponenty. Chociaż pliki konfiguracyjne można przygotowywać w dowolnym edytorze tekstu, dużo lepszym rozwiązaniem jest użycie narzędzi specjalnie zaprojektowanych z myślą o tym zadaniu (np. Framework Configuration). Pliki konfiguracyjne będziemy wykorzystywali przede wszystkim do testowania aplikacji

³ Patrz strona internetowa <http://msdn.microsoft.com/netframework/downloads/updates/default.aspx>.

pod kątem ich zgodności z nowymi wersjami platformy .NET. Chociaż teoretycznie jest to możliwe, uruchamianie aplikacji w wersjach starszych od tej, w której tę aplikację skompilowano, zwykle nie ma sensu.

Narzędzia platformy .NET Framework

Platforma .NET Framework automatyzuje możliwie dużą część zadań i w wielu przypadkach skutecznie ukrywa przed programistą szczegóły stosowanych rozwiązań. Istnieją jednak sytuacje, w których interwencja programisty jest absolutnie konieczna. Odpowiednie działania mogą wymagać doskonałej znajomości szczegółów funkcjonowania zestawu oraz doświadczenia w przygotowywaniu aplikacji do wdrożenia. Podczas pracy z technologią .NET odkryliśmy wiele przykładów tego rodzaju zadań. Udział programisty będzie nieodzowny podczas realizacji następujących zadań:

- dodawania pliku do zestawu,
- przeglądania zawartości zestawu,
- przeglądania szczegółowych informacji o konkretnej klasie,
- generowania pary kluczy publicznych i prywatnych celem utworzenia silnie nazwanego zestawu,
- edycji plików konfiguracyjnych.

Wiele z tych zadań bardziej lub mniej szczegółowo omówimy w kolejnych rozdziałach. Warto jednak mieć świadomość istnienia narzędzi, które ułatwiają wykonywanie odpowiednich działań — sztukę efektywnego korzystania z części tych narzędzi (szczególnie tych wspierających przeglądanie klas i zestawów) należy opanować już na wczesnych etapach poznawania technologii .NET.

W tabeli 1.3 wymieniono i krótko opisano kilka najbardziej przydatnych narzędzi platformy .NET Framework, które ułatwiają wytwarzanie i wdrażanie aplikacji. Trzy narzędzia, *Ildasm.exe*, *wincv.exe* i .NET Framework Configuration szczegółowo omówimy w dalszej części tego rozdziału.

Wiele spośród opisanych powyżej katalogów znajduje się w następującym podkatalogu zestawu narzędzi .NET Framework SDK:

```
c:\Program Files\Microsoft.NET\SDK\v2.0\Bin
```

Uruchamianie wymienionych przed chwilą narzędzi w wierszu poleceń systemu operacyjnego Windows (z dowolnego katalogu) wymaga w pierwszej kolejności dopisania ścieżki do odpowiednich plików wykonywalnych w zmiennej systemowej *Path*. Aby to zrobić, należy wykonać następujące kroki:

1. Kliknij prawym przyciskiem myszy ikonę *Mój komputer* i z wyświetlonego menu kontekstowego wybierz opcję *Właściwości*.
2. Kliknij zakładkę *Zaawansowane* i przycisk *Zmienne środowiskowe*.
3. Zaznacz zmienną *Path* i dodaj do niej ścieżkę do odpowiedniego podkatalogu zestawu narzędzi SDK.

Tabela 1.3. Wybrane narzędzia platformy .NET Framework

Narzędzie	Opis
<i>Al.exe</i> Assembly Linker	Narzędzia Assembly Linker można używać do tworzenia zestawów złożonych z modułów wygenerowanych przez różne kompilatory. Assembly Linker jest wykorzystywany także do budowy zestawów obejmujących same zasoby (tzw. zestawów satelickich).
<i>Fuslogvw.exe</i> Assembly Binding Log Viewer	Narzędzie Assembly Binding Log Viewer jest wykorzystywane najczęściej wtedy, gdy w procesie wczytywania zestawów wystąpią jakieś problemy. Pozwala skutecznie śledzić kroki składające się na próby wczytania zestawu.
<i>Gacutil.exe</i> Narzędzie Global Assembly Cache	Narzędzie jest wykorzystywane do instalowania lub usuwania zestawów z globalnej pamięci podręcznej zestawów (GAC).
<i>Ildasm.exe</i> MSIL Disassembler	Narzędzie MSIL Disassembler służy do przeglądania zestawów, w tym kodu języka pośredniego (IL) oraz metadanych.
<i>Mscorcfg.msc</i> Narzędzie .NET Framework Configuration	Dołączone narzędzie Microsoft Management Console (MMC) zaprojektowano z myślą o konfigurowaniu zestawów bez konieczności samodzielnego wprowadzania zmian w plikach konfiguracyjnych aplikacji. Dodatkowy składnik tego narzędzia, Framework Wizards, stworzono przede wszystkim dla administratorów, choć jest wykorzystywany także przez programistów.
<i>Ngen.exe</i> Native Image Generator	Kompiluje język pośredni zestawu do postaci odpowiedniego kodu maszynowego. Generowany w ten sposób obraz jest następnie umieszczany w pamięci podręcznej rdzennych obrazów.
<i>Sn.exe</i> Narzędzie Strong Name	Generuje klucze, które są następnie wykorzystywane do tworzenia silnie nazwanych (podpisanych) zestawów.
<i>wincv.exe</i> Windows Forms Class Viewer	Interfejs graficzny wyświetlający i umożliwiający przeszukiwanie informacji o klasie.
<i>Wsd.exe</i> Narzędzie Web Services Description Language	Generuje opisowe informacje o usłudze sieciowej (Web Service) — informacje mogą być wykorzystywane przez klienta uzyskującego dostęp do danej usługi.

Jeśli masz zainstalowane środowisko Visual Studio, dużo prostszym rozwiązaniem będzie użycie prekonfigurowanego wiersza poleceń tego środowiska. Wiersz poleceń środowiska VS.NET automatycznie inicjalizuje informacje o ścieżkach, które są niezbędne do uzyskiwania dostępu do narzędzi wiersza poleceń.

Ildasm.exe

Narzędzie *Intermediate Language Disassembler* (Ildasm), które jest częścią zestawu narzędzi .NET Framework SDK, zwykle znajduje się w podkatalogu *Bin* w katalogu, w którym zainstalowano ten zestaw narzędzi. Narzędzie Ildasm jest bezcenną pomocą podczas analizy środowiska zestawów platformy .NET i jako takie będzie jednym z pierwszych narzędzi, z którym będziesz się musiał zaznajomić już na początku swojej przygody z zestawami .NET.

Najprostszym sposobem użycia narzędzia Intermediate Language Disassembler jest wpisanie w wierszu poleceń następującego wyrażenia:

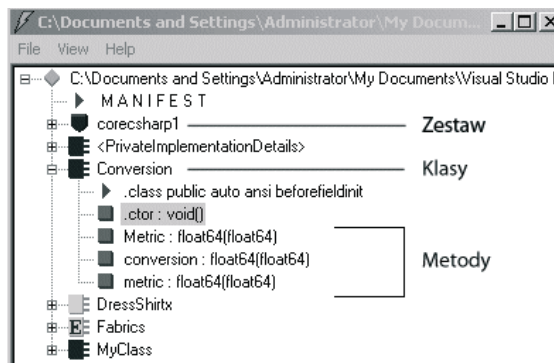
```
C:\>Ildasm /adv
```

Opcjonalny przełącznik */adv* powoduje, że będą dostępne zaawansowane opcje wyświetlania. Wykonanie tego polecenia wywoła graficzny interfejs użytkownika (GUI) z menu File, w którym będziemy mogli wybrać interesujący nas zestaw. Warto pamiętać, że narzędzie Ildasm nie otwiera plików składowanych w pamięci podręcznej zestawów (GAC).

Na rysunku 1.9 przedstawiono przykład danych wyjściowych wyświetlonych przez narzędzie Ildasm po wybraniu jednego z zestawów. Zawartość zestawu jest prezentowana w czytelnym, hierarchicznym formacie zawierającym nazwę zestawu, w tym przypadku *corecsharp1*, oraz wszystkie jego składniki.

Rysunek 1.9.

Zawartość
wybranego zestawu
wyświetlona
przez narzędzie
Ildasm.exe



Użytkownik narzędzia może wejść w głąb tej hierarchii, aby zapoznać się z reprezentowanym przez poszczególne składniki kodem języka pośredniego IL (lub wspólnego języka pośredniego — CIL). Z rysunku 1.9 wynika, że zestaw *corecsharp1* składa się z trzech metod: *Metric*, *conversion* i *metric*. Potwierdza to oryginalny kod klasy *Conversion* napisany w języku C#:

```
public class Conversion
{
    public double Metric( double inches)
    { return (2.54 * inches); }
    [CLSCompliantAttribute(false)]
    public double metric( double miles)
    { return (miles / 0.62); }
    public double conversion( double pounds)
    { return (pounds * 454); }
}
```

Dwukrotne kliknięcie metody *Metric* spowoduje wyświetlenie ekranu z kodem języka pośredniego (IL) tej metody — patrz rysunek 1.10.

Narzędzie Ildasm może być wykorzystywane do nauki i eksperymentowania z koncepcją języka pośredniego i zestawów. Program znajduje też ważne zastosowania praktyczne. Przypuśćmy, że dysponujemy komponentem (zestawem .NET) opracowanym przez kogoś

Rysunek 1.10.

Widok kodu języka pośredniego (IL)

```

Conversion::Metric : float64(float64)
.method public hidebysig instance float64
    Metric(float64 inches) cil managed
{
    // Code size      16 (0x10)
    .maxstack 2
    .locals init (float64 V_0)
    IL_0000: ldc.r8      2.54
    IL_0001: ldarg.1
    IL_0002: mul
    IL_0003: stloc.0
    IL_0004: br.s      IL_000e
    IL_0005: ldloc.0
    IL_0006: ret
} // end of method Conversion::Metric
  
```

innego, i że nie posiadamy żadnej dokumentacji na jego temat. W takich przypadkach poszukiwanie szczegółów na temat interfejsu tego zestawu powinniśmy rozpocząć właśnie od jego otwarcia w narzędziu Ildasm.

Narzędzie Ildasm zawiera w menu *File* opcję *Dump*, za pomocą której możemy zapisać dokumentację programu w pliku tekstowym. Aby utworzyć długi, ale czytelny dokument o metadanych danego zestawu, wybierz opcję *Dump Metainfo*; aby zapoznać się z profilem zestawu wyłącznie z liczbą bajtów wykorzystywanych przez poszczególne składniki, wybierz opcję *Dump Statistics*.

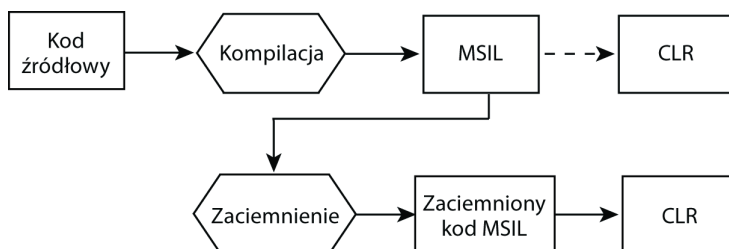
Narzędzie Ildasm i zaciemnianie kodu

Jednym z naturalnych odruchów programistów aplikacji .NET jest pytanie o sposób ochrony ich własności intelektualnej w sytuacji, gdy istnieją takie narzędzia jak Ildasm (i inne komercyjne programy odwracające kompilację), które mogą być wykorzystywane do odtworzenia oryginalnego kodu na podstawie skompilowanych zestawów. Jednym z rozwiązań jest technika nazywana **zaciemnianiem kodu** (ang. *obfuscation*), która polega na zręcznym modyfikowaniu nazw i innych operacjach na kodzie skutecznie uniemożliwiających jego odczytanie przez ludzi.

Należy pamiętać, że zaciemnianie kodu nie jest tożsame z jego szyfrowaniem. Szyfrowanie wymagałoby wykonywania dodatkowego kroku odszyfrowywania przed przetworzeniem kodu przez kompilator JIT. Zaciemnianie polega na transformacji kodu języka pośredniego do postaci, w której można go skompilować za pomocą standardowych narzędzi środowiska wytwarzania oprogramowania (patrz rysunek 1.11).

Rysunek 1.11.

Zaciemnianie kodu skutecznie ukrywa oryginalny kod języka pośredniego (IL)



Zaciemniony kod funkcjonalnie nie odbiega od oryginalnego kodu języka pośredniego danego zestawu i podczas wykonywania we wspólnym środowisku uruchomieniowym (CLR) generuje identyczne wyniki. Jak to możliwe? Otóż typowym działaniem w ramach zaciemniania kodu jest zamiana czytelnych nazw typów i składowych na nazwy, które czytelnikowi takiego kodu niczego nie mówią. Przykładowo, kiedy zajrzysz do zaciemnionego kodu, znajdziesz tam mnóstwo typów nazwanych a lub b. Algorytm zaciemniania musi oczywiście być na tyle „inteligentny”, by nie zmieniać nazw typów wykorzystywanych poza zmodyfikowanym zestawem, ponieważ każda taka zmiana uniemożliwiłaby prawidłowe odwołania do tych typów. Innym typowym rozwiązaniem jest modyfikowanie przepływu sterowania (ang. *control flow*) w sposób, który nie zmienia logiki działania programu. Przykładowo, wyrażenie `while` można zastąpić kombinacją wyrażzeń `goto` oraz `if`.

Zestaw narzędzi .NET Framework SDK nie zawiera własnego mechanizmu zaciemniania kodu. Środowisko programowania Visual Studio .NET jest oferowane wraz z narzędziem Dotfuscator Community Edition, czyli wersją komercyjnego produktu Dotfuscator z nieco ograniczoną funkcjonalnością. Chociaż wspomniane narzędzie nie jest specjalnie wyrafinowane i działa wyłącznie w środowiskach firmy Microsoft, warto właśnie od niego rozpocząć poznawanie tego ciekawego procesu. Wielu producentów oferuje bardziej zaawansowane produkty w tym zakresie.

WinCV.exe

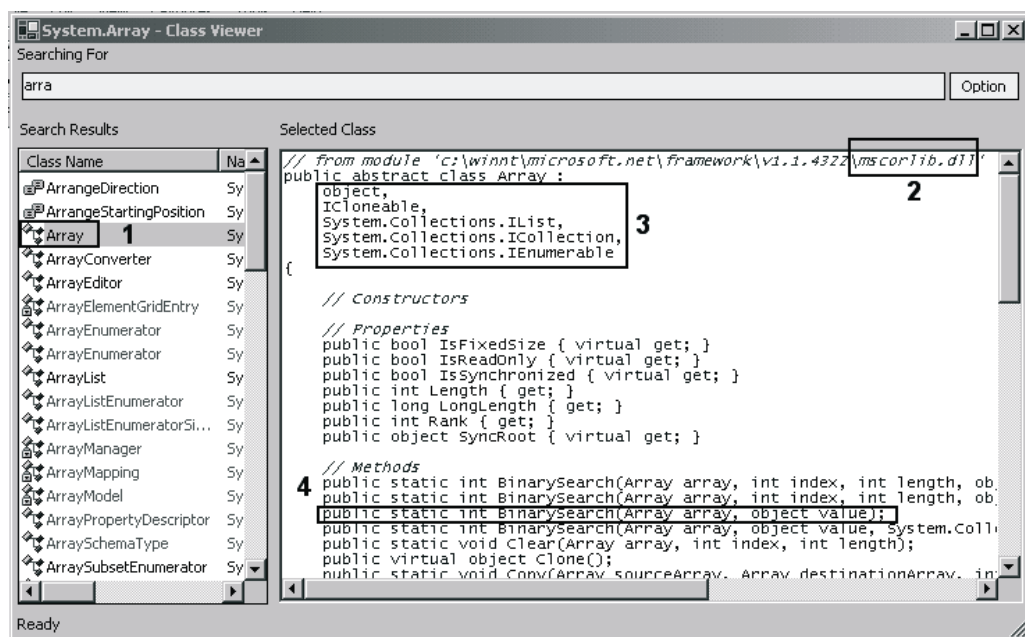
WinCV jest przeglądarką klas analogiczną do narzędzia Visual Studio Object Viewer oferowaną w ramach środowiska VS .NET. Program znajduje się w katalogu *Program Files\Microsoft.NET\SDK\VL.x\Bin* i można go uruchamiać z poziomu wiersza poleceń. Po wyświetleniu okna graficznego interfejsu użytkownika WinCV należy wpisać nazwę interesującej nas klasy w polu tekstowym *Searching For* (patrz rysunek 1.12).

WinCV oferuje bogaty zasób informacji o wszystkich typach obsługiwanych przez bibliotekę klas bazowych. Cztery obszary zaznaczone na rysunku 1.12 pokazują próbki dostępnych danych:

1. Przedmiotem analizy jest klasa bazowa `System.Array`.
2. Klasa `System.Array` jest częścią zestawu *mscorlib.dll*. Wspominaliśmy już, że zestaw *mscorlib.dll* zawiera najważniejsze typy zarządzane platformy .NET.
3. Lista zawiera klasę, obiekt i interfejsy dziedziczone przez klasę `System.Array`.
4. Wyświetlona reprezentacja zawiera definicję każdej z metod tej klasy, która obejmuje uprawnienia dostępu, typ oraz parametry — definicja w tej postaci jest nazywana **sygnaturą metody**.

Narzędzie Framework Configuration

Narzędzie Framework Configuration oferuje proste rozwiązania w zakresie zarządzania i konfiguracji zestawów oraz ustawiania zabezpieczeń w dostępie do kodu. Narzędzie jest częścią konsoli *Microsoft Management Console* (MMC). Aby je uruchomić, w *Panelu sterowania*



Rysunek 1.12. Przykład użycia narzędzia WinCV do przeglądania definicji typu klasy Array

kliknij ikonę *Narzędzia administracyjne* i wybierz skrót *Microsoft .NET Framework Konfiguracja*. Program Framework Configuration stworzono z myślą o administratorach, którzy muszą wykonywać następujące zadania:

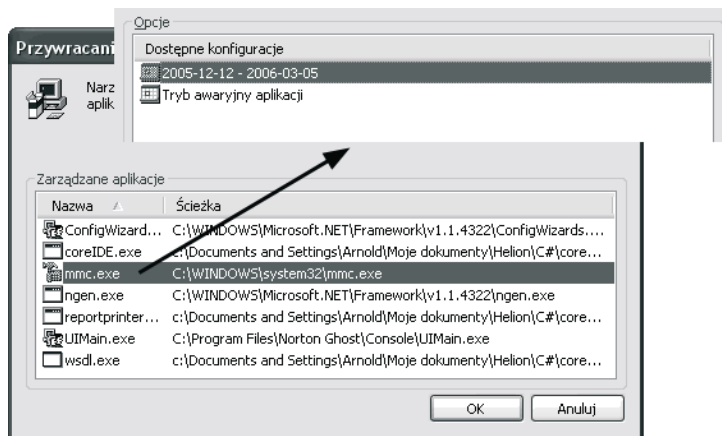
- **Zarządzanie zestawami.** Zestawy mogą być dodawane lub usuwane z pamięci podręcznej GAC.
- **Konfigurowanie zestawów.** W razie aktualizacji zestawu osoba, która go opublikowała, jest odpowiedzialna za odpowiednie dostosowanie **reguł wiązania** (ang. *binding policy*) tego zestawu. Na podstawie tych reguł wspólne środowisko uruchomieniowe określa właściwą wersję wczytywanego zestawu w odpowiedzi na wykryte odwołanie do zestawu w którejś z wykonywanych aplikacji. Przykładowo, jeśli zestaw w wersji 1.1 zastąpi zestaw 1.0, reguły dołączania powinny tak przekierować odwołania do wersji 1.0, aby środowisko CLR wczytywało wersję 1.1. Odpowiednie informacje należy umieścić w pliku konfiguracyjnym.
- **Analiza zabezpieczeń platformy .NET Framework i modyfikowanie reguł bezpieczeństwa zestawów.** Mechanizmy zabezpieczeń platformy .NET umożliwiają przypisywanie poszczególnym zestawom określonych uprawnień dostępu. Co więcej, docelowy zestaw może wymagać posiadania uprawnień dostępu od pozostałych zestawów, które żądają dostępu do danego zestawu.
- **Zarządzanie interakcją poszczególnych aplikacji z zestawem lub zbiorem zestawów.** Narzędzie Framework Configuration umożliwia przeglądanie nie tylko list wszystkich zestawów wykorzystywanych przez daną aplikację, ale też kompletnych zbiorów wykorzystywanych wersji.

Aby lepiej zilustrować praktyczne zastosowania tego narzędzia konfiguracyjnego, przeanalizujemy sposób jego użycia do rozwiązania jednego z najczęstszych problemów napotykanym w procesie wytwarzania oprogramowania — konieczności powrotu do ostatniej działającej wersji w razie problemów z funkcjonowaniem wersji bieżącej. Realizacja tego zadania może być szczególnie trudna w sytuacji, gdy aplikacja wykorzystuje po stronie serwera biblioteki DLL lub zestawy. Okazuje się, że twórcom technologii .NET udało się znaleźć dość sprytnie rozwiązanie — za każdym razem, gdy uruchamiamy jakąś aplikację, środowisko uruchomieniowe zapisuje w dzienniku zdarzeń informacje o wszystkich wykorzystywanych zestawach. Jeśli wersje tych zestawów nie zmieniają się od ostatniego uruchomienia, zostaną zignorowane przez środowisko CLR; w przeciwnym razie zostanie zachowana „migawka” nowego zbioru zestawów.

Kiedy próba uruchomienia aplikacji zakończy się niepowodzeniem lub aplikacja nie będzie działała prawidłowo, programista będzie mógł w prosty sposób powrócić do wersji zestawów, które nie powodowały podobnych problemów. Narzędzie konfiguracyjne może posłużyć do przekierowania aplikacji do wcześniejszej wersji zaktualizowanego ostatnio zestawu. Przydatność tego rodzaju narzędzi jest szczególnie widoczna w sytuacji, gdy ewentualny problem dotyczy więcej niż jednego zestawu — programista może się zapoznać z dotychczasowymi konfiguracjami zestawów i wybrać całą zestaw zestawów dla danej aplikacji.

Aby wyświetlić i ewentualnie wybrać poprzednie konfiguracje, w menu narzędzia *Framework Configuration* kliknij kolejno pozycję *Applications* i opcję *Fix an Application*. Na rysunku 1.13 przedstawiono fragmenty obu wyświetlonych okien dialogowych. Okno główne zawiera listę aplikacji, których wykonanie zostało zarejestrowane w dzienniku zdarzeń. Mniejsze okno (w tym przypadku obcięta część większego okna dialogowego) zostanie wyświetlone po kliknięciu którejs z tych aplikacji. Okno zawiera listę ostatnich konfiguracji (maksymalnie pięciu) przypisanych do danej aplikacji. Wystarczy wybrać konfigurację zestawów, której chcemy użyć dla wskazanej aplikacji.

Rysunek 1.13.
Przykład użycia narzędzia konfiguracyjnego aplikacji do ustawiania wersji zestawów



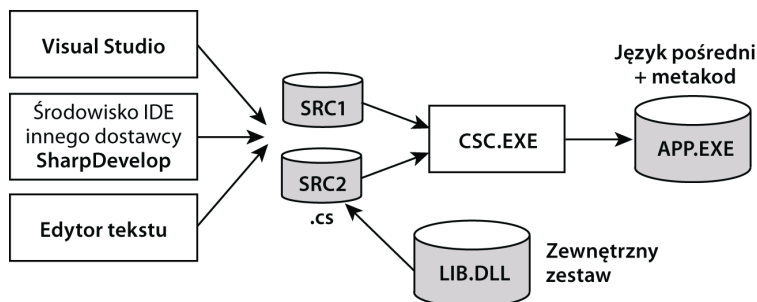
Jak widać, narzędzie konfiguracyjne stworzono przede wszystkim z myślą o administratorach. Programiści powinni korzystać tylko z wybranych podzbiorów funkcjonalności tego narzędzia zawartych w trzech kreatorach: *Dopasuj zabezpieczenia .NET*, *Ufaj zestawowi* oraz *Napraw aplikację*. Dostęp do tych kreatorów można uzyskać, klikając ikonę *Microsoft .NET Framework Kreatorzy* w *Narzędziach administracyjnych*.

1.5. Wyjaśnienie działania kompilatora C#

Wielu programistów piszących złożone aplikacje dla platformy .NET wykorzystuje Visual Studio lub jakieś inne zintegrowane środowisko wytwarzania oprogramowania (ang. *Integrated Development Environment* — *IDE*) do edycji kodu, dołączania zewnętrznych zestawów, diagnozowania aplikacji i generowania ostatecznych, skompilowanych wersji oprogramowania. Jeśli i Ty zaliczasz się do tej grupy, znajomość technik korzystania z zestawu narzędzi .NET SDK i podstawowego kompilatora C# najprawdopodobniej nie będzie Ci potrzebna, choć z pewnością podniosłaby poziom wiedzy o procesie kompilacji w technologii .NET i dała lepsze podstawy do pracy z zestawami. Efektem ubocznym korzystania z kompilatora C# na poziomie wiersza poleceń byłoby też zdobycie cennego doświadczenia w korzystaniu z programów wchodzących w skład zestawu narzędzi SDK. Wiele spośród narzędzi zaprezentowanych w poprzednim podrozdziale jest wywoływanych właśnie w wierszu poleceń, a w niektórych sytuacjach możliwość przeprowadzania procesu kompilacji w takim środowisku okazuje się kuszącą alternatywą dla zaawansowanych mechanizmów środowisk IDE.

Na rysunku 1.14 przedstawiono podstawowe kroki składające się na proces konwertowania kodu źródłowego na ostateczną, skompilowaną wersję oprogramowania. Celem niniejszego podrozdziału jest prezentacja, jak edytor tekstu w połączeniu z kompilatorem C# może być wykorzystywany konstruowania aplikacji. Przy okazji przeanalizujemy wiele opcji kompilatora, które często pozostają ukryte przed użytkownikami zintegrowanych środowisk wytwarzania oprogramowania (IDE).

Rysunek 1.14.
Schemat procesu kompilacji



Lokalizowanie kompilatora

Kompilator języka programowania C#, *csc.exe*, znajduje się w katalogu, w którym zainstalowano platformę .NET Framework:

```
C:\winnt\Microsoft.NET\Framework\v2.0.40607
```

Ścieżka do pliku wykonywalnego kompilatora może się oczywiście różnić w zależności od stosowanego systemu operacyjnego i zainstalowanej wersji platformy .NET Framework. Aby kompilator C# był dostępny w wierszu poleceń niezależnie od bieżącego katalogu,

powinieneś dodać tę ścieżkę do zmiennej systemowej Path. Odpowiednią procedurę opisano w poprzednim podrozdziale przy okazji omawiania ścieżek dostępu do programów wchodzących w skład zestawu narzędzi SDK.

Aby się upewnić, że kompilator rzeczywiście jest dostępny z dowolnego katalogu, wpisz w wierszu poleceń następujące wyrażenie:

```
C:\>csc /help
```

Kompilowanie oprogramowania z poziomu wiersza poleceń

Aby skompilować napisaną w języku C# aplikację konsoli, nazwaną *client.cs*, do postaci pliku wykonywalnego *client.exe*, należy wykonać na poziomie wiersza poleceń jedno z przedstawionych poleceń:

```
C:\> csc client.cs
C:\> csc /t:exe client.cs
```

Oba wyrażenia spowodują, że plik z kodem źródłowym zostanie skompilowany do postaci pliku wykonywalnego z rozszerzeniem *.exe*, czyli domyślnego formatu wyjściowego kompilatora. Typ docelowy można określić za pomocą flagi */t:* (patrz tabela 1.4). Aby utworzyć plik DLL, należy ustawić dla tej flagi wartość *library*, natomiast aplikacja WinForms będzie wymagała użycia flagi */t:winexe*. Warto pamiętać, że jeśli tworząc aplikację WinForms, użyjemy flagi */t:exe*, kompilacja zakończy się pomyślnie, tyle że w czasie wykonywania aplikacji zostanie wyświetlone w tle okno konsoli.

Największą zaletą korzystania z podstawowego kompilatora języka C# (wywoływanego z poziomu wiersza poleceń) jest możliwość pracy z wieloma plikami i zestawami. Aby lepiej to zilustrować, napiszemy w języku C# dwa proste pliki źródłowe: *client.cs* i *clientlib.cs*.

client.cs

```
using System;
public class MyApp
{
    static void Main(string[] args)
    {
        ShowName.ShowMe("Core C#");
    }
}
```

clientlib.cs

```
using System;
public class ShowName
{
    public static void ShowMe(string MyName)
    {
        Console.WriteLine(MyName);
    }
}
```


Tabela 1.4. Wybrane opcje kompilatora C# uruchamianego w wierszu poleceń

Opcja	Opis
/addmodule	Określa moduł, który należy dołączyć do tworzonego zestawu. Jest to najprostszy sposób tworzenia wieloplukowych zestawów.
/debug	Wymusza wygenerowanie informacji diagnostycznych.
/define	Umożliwia przekazywanie do kompilatora dyrektywy preprocesora: /define:DEBUG.
/delaysign	Kompiluje zestaw z silną nazwą, korzystając z techniki odroczonego podpisywania (ang. <i>delayed signing</i> — patrz rozdział 15.).
/doc	Służy do wskazywania pliku wyjściowego dla generowanej dokumentacji XML.
/keyfile	Określa ścieżkę do pliku <i>.snk</i> zawierającego parę kluczy niezbędną do silnego nazwania danego zestawu (patrz rozdział 15.).
/lib	Określa miejsce składowania dołączanych zestawów zewnętrznych (wskazanych w opcji /reference).
/out	Określa nazwę docelowego pliku dla skompilowanej wersji. Nazwa domyślna jest taka sama jak dla pliku wejściowego (rozszerzenie <i>.cs</i> jest zastępowane rozszerzeniem <i>.exe</i>).
/reference (/r)	Odwołuje się do zewnętrznego zestawu.
/resource	Służy do osadzania plików zasobów w ramach tworzonego zestawu.
/target (/t)	Określa typ tworzonego pliku wyjściowego: /t:exe kompiluje aplikację konsoli nazwaną <i>*.exe</i> . Flaga /t:exe jest domyślną opcją kompilacji. /t:library kompiluje zestaw nazwany <i>*.dll</i> . /t:module kompiluje moduł (przenośny plik wykonywalny), który nie zawiera manifestu. /t:winexe kompiluje zestaw Windows Forms nazwany <i>*.exe</i> .

Na tym etapie nie ma znaczenia, czy rozumiesz szczegółowe konstrukcje językowe — wystarczy, że będziesz wiedział, że kod zawarty w pliku *client.cs* wywołuje funkcję zdefiniowaną w pliku *clientlib.cs* i wyświetla otrzymaną wartość w oknie konsoli. Kompilator C# oferuje wiele sposobów implementacji tego rodzaju związków, które demonstrują nie tylko podstawowe opcje kompilatora, ale też rzucają nieco światła na ogólną koncepcję zestawów.

Przykład 1.: kompilacja wielu plików

Kompilator języka C# może otrzymać na wejściu dowolną liczbę plików źródłowych. Wygenerowane dane wyjściowe zostaną umieszczone w pojedynczym pliku zestawu:

```
csc /out:client.exe client.cs clientlib.cs
```

Przykład 2.: tworzenie i używanie biblioteki klas

Skompilowany kod z pliku źródłowego *clientlib.cs* można umieścić w osobnej bibliotece, która będzie następnie udostępniana wszystkim zainteresowanym klientom:

```
csc /t:library clientlib.cs
```

W wyniku tej operacji otrzymamy zestaw nazwany *clientlib.dll*. Możemy teraz skompilować kod klienta, stosując polecenie odwołujące się do tego zewnętrznego zestawu:

```
csc /r:clientlib.dll client.cs
```

Kompilator wygeneruje zestaw nazwany *client.exe*. Jeśli przeanalizujesz ten zestaw za pomocą narzędzia *Ildasm*, przekonasz się, że jego manifest zawiera odwołanie do skompilowanego wcześniej zestawu *clientlib.dll*.

Przykład 3.: tworzenie zestawu wieloplikowego

Zamiast tworzyć osobny zestaw, skompilowaną wersję kodu zawartego w pliku *clientlib.cs* można umieścić wewnątrz zestawu *client.exe*. Ponieważ tylko plik w zestawie może zawierać manifest, w pierwszej kolejności będziemy musieli skompilować plik *clientlib.cs* do postaci **przenośnego modułu wykonywalnego**⁴ (ang. *Portable Executable* — *PE*). Aby otrzymać taki moduł, wystarczy w wywołaniu kompilatora użyć opcji `/t:module`:

```
csc /t:module clientlib.cs
```

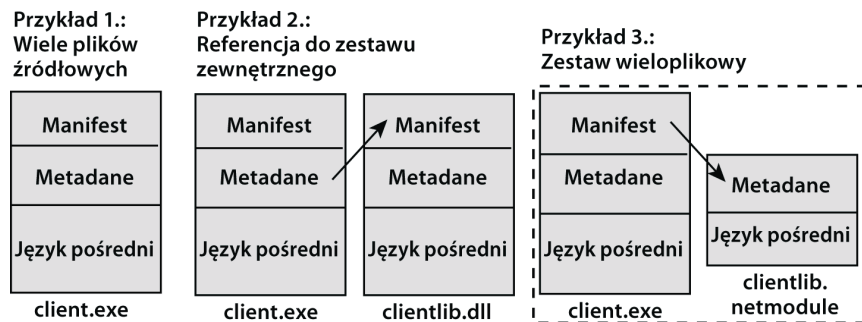
Plik wyjściowy zostanie nazwany *clientlib.netmodule*. Można ten plik umieścić w zestawie *client.exe* za pomocą przełącznika `/addmodule` kompilatora *csc.exe*:

```
csc /addmodule:clientlib.netmodule client.cs
```

Wynikowy zestaw będzie się składał z dwóch plików: *client.exe* i *clientlib.netmodule*.

Przykłady przedstawione na rysunku 1.15 dobrze pokazują, że nawet w przypadku tak prostej aplikacji programista ma do dyspozycji szereg rozwiązań i musi podjąć właściwą decyzję odnośnie architektury implementowanego oprogramowania.

Rysunek 1.15.
Możliwe rozwiązania w zakresie wdrażania aplikacji



⁴ Format PE definiuje układ plików wykonywalnych, które można uruchamiać w 32- i 64-bitowych systemach Windows.

1.6. Podsumowanie

Platforma .NET Framework składa się ze wspólnego środowiska uruchomieniowego (CLR) i biblioteki klas platformy (FCL). Środowisko CLR zarządza wszystkimi zadaniami związanymi z wykonywaniem kodu aplikacji. Pierwszym zadaniem tego środowiska jest sprawdzenie, czy uruchamiana aplikacja rzeczywiście jest zgodna ze standardową specyfikacją wspólnego języka (CLS). Bezpośrednio potem wspólne środowisko uruchomieniowe wczytuje aplikację i lokalizuje wszystkie zależne zestawy. Wykorzystywany przez to środowisko kompilator JIT (*Just-in-Time*) konwertuje kod języka pośredniego zawarty w zestawie aplikacji (najmniejszej jednostce wdrożeniowej w technologii .NET) do postaci rdzennego kodu maszynowego. W czasie właściwego wykonywania programu środowisko CLR odpowiada za zapewnianie bezpieczeństwa, zarządzanie wątkami, przydział pamięci oraz zwalnianie pamięci nieużywanej (z wykorzystaniem odpowiedniego mechanizmu odzyskiwania pamięci).

Cały kod musi być umieszczony w specjalnych zestawach, które mogą być prawidłowo obsługiwane przez wspólne środowisko uruchomieniowe (CLR). Zestaw może mieć albo postać pojedynczego pliku, albo grupy wielu fizycznych plików traktowanych jak pojedyncza jednostka wdrożeniowa. Zestaw może zawierać zarówno moduły kodu, jak i niezbędne pliki zasobów.

Biblioteka FCL zawiera zbiór klas i pozostałych typów wielokrotnego użytku, które mogą być wykorzystywane przez dowolny zestaw zgodny z wymaganiami wspólnego środowiska uruchomieniowego (CLR). Takie rozwiązanie eliminuje konieczność stosowania bibliotek właściwych dla poszczególnych kompilatorów. Chociaż biblioteka FCL składa się z wielu fizycznych plików DLL zawierających ponad tysiąc różnych typów, przeszukiwanie tej biblioteki jest stosunkowo proste dzięki zastosowaniu przestrzeni nazw, które tworzą logiczną hierarchię obejmującą wszystkie obsługiwane typy.

Platforma .NET Framework zawiera szereg przydatnych narzędzi, które mogą pomóc nie tylko programiście w diagnozowaniu i wdrażaniu oprogramowania, ale także administratorowi w realizacji takich zadań jak zarządzanie zestawami, prekompilowanie zestawów, dodawanie nowych plików do zestawów oraz przeglądanie szczegółowych informacji o klasach. Co więcej, istnieje bogaty zbiór narzędzi typu open source (oferowanych z otwartym dostępem do kodu źródłowego), które także mogą stanowić cenną pomoc w procesie wytwarzania oprogramowania dla platformy .NET.

1.7. Sprawdź, czego się nauczyłeś

1. Jakie przenośne środowisko należy zainstalować na komputerze klienta (użytkownika), aby można na nim było uruchomić aplikację .NET?
2. Czym jest zarządzany, a czym jest kod niezarządzany?
3. Jaka jest różnica pomiędzy wspólnym systemem typów (CTS) a specyfikacją wspólnego języka (CLS)?

- 4.** Jak to możliwe, że we wspólnym środowisku uruchomieniowym kod wygenerowany przez różne kompilatory może ze sobą współpracować?
- 5.** Jaka rolę pełni globalna pamięć podręczna zestawów?
- 6.** Wymień cztery składniki, które składają się na identyfikator silnie nazwanego zestawu.
- 7.** Jaka relacja łączy przestrzeń nazw z zestawem .NET?
- 8.** Rozwiń i krótko opisz pojęcia kryjące się za następującymi akronimami: CLR, GAC, FCL, IL.